

# Theoretical Foundations

Concepts and Theorems in Distributed  
Computing

Walter Kriha

# Goals

The goal is to give students a sound theoretical foundation to understand complex distributed systems. These systems differ significantly from local forms of computing.

The influence of DS theorems can be subtle but there is no large scale distributed design without those.

# Overview

## Basic Concepts

Distributed Systems Fallacies

Latency

Correctness and Liveness

Time, Ordering and Failures

The Impossibility of Consensus in Async. DS (FLP)

The CAP Theorem

Failures, Failure Types and Failure Detectors

Time in Distributed Systems

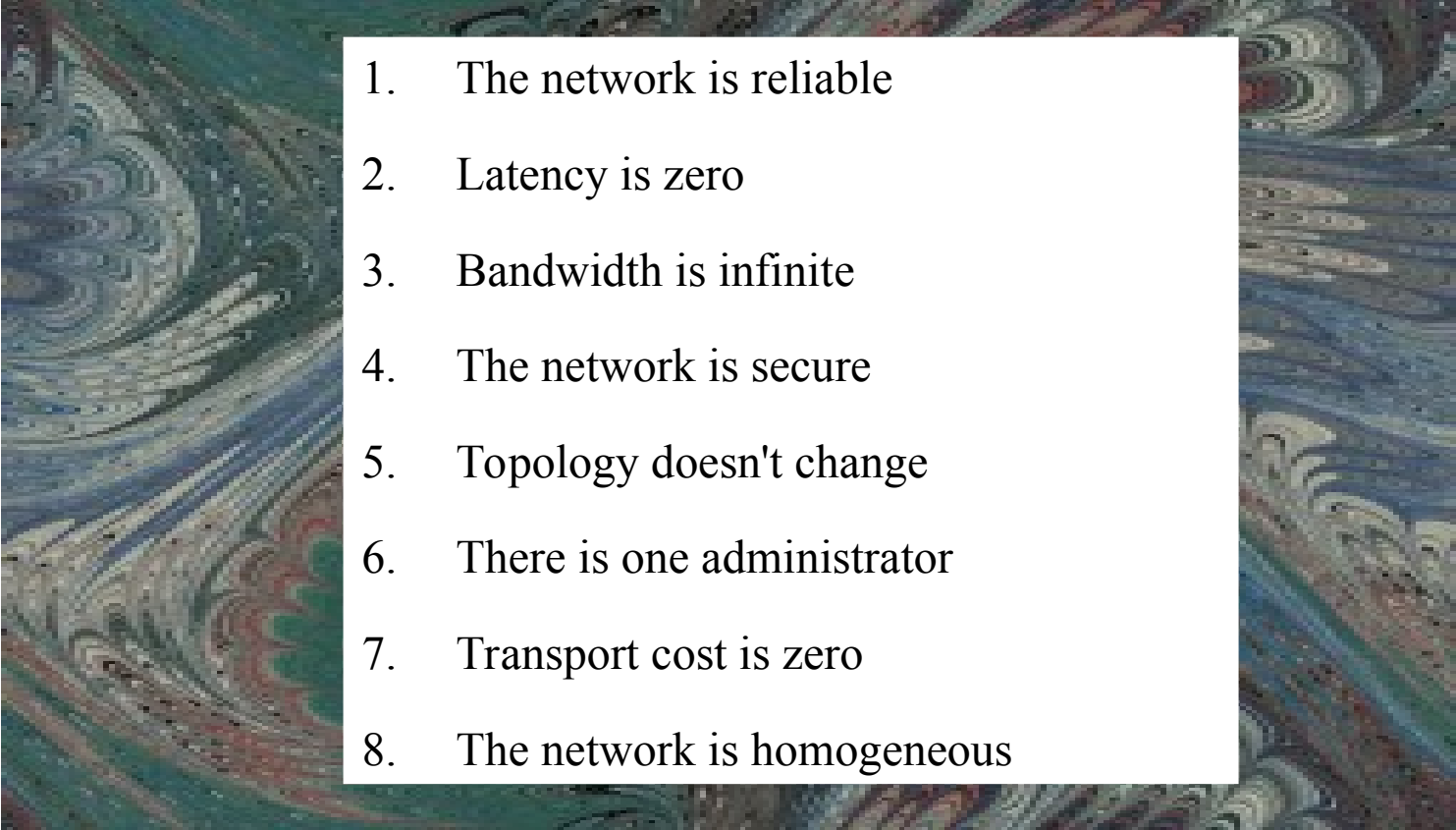
Ordering and Causality

Algorithms for Consensus in DS

Optimistic Replication and Eventual Consistency

# Basic Concepts

# The Eight Fallacies of Distributed Computing

- 
1. The network is reliable
  2. Latency is zero
  3. Bandwidth is infinite
  4. The network is secure
  5. Topology doesn't change
  6. There is one administrator
  7. Transport cost is zero
  8. The network is homogeneous

“Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.” (see also: Fallacies of Distributed Computing Explained, Arnon Rotem-Gal-Oz.) The 9<sup>th</sup> fallacy: ignoring usage costs when moving to the cloud! Moving data in the cloud is costly!

# Example: Reliable Network

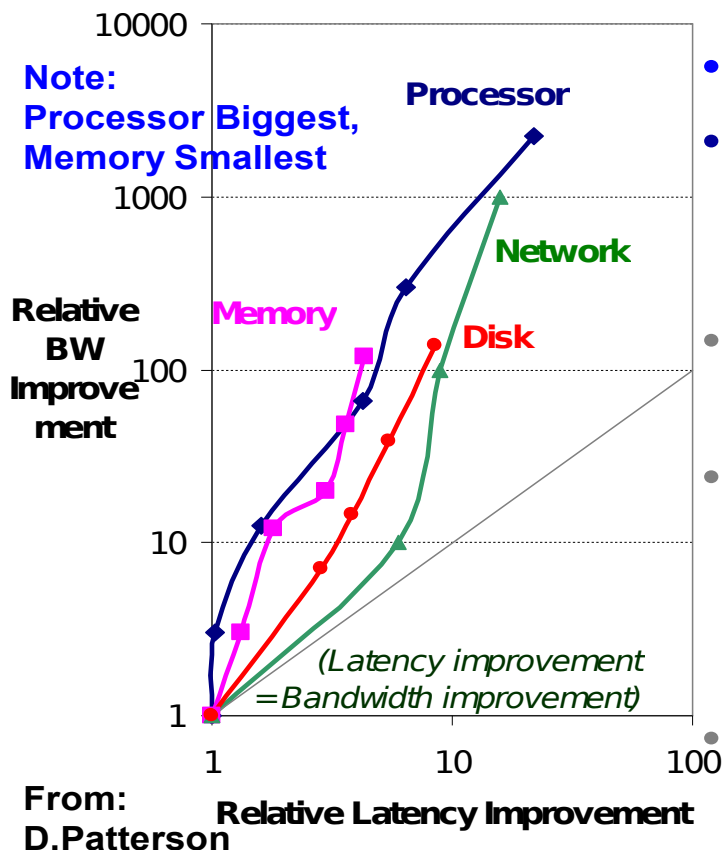


From Kamille Fournie's talk at Strangeloop 2015

# Example: Latency

1. Know the long term trends in hardware
2. Understand the problem of deep queuing networks and the solutions
3. Know your numbers with respect to switching times, router delays, round-trip times, IOPS per devices and perform “back of the envelope” calculations
4. Understand buffering effects on latency
5. Include the client side in your calculations

# Latency Lags Bandwidth (last ~20 years)



- Performance Milestones

- Processor: '286, '386, '486, Pentium, Pentium Pro, Pentium 4 (21x,2250x)

- Ethernet: 10Mb, 100Mb, 1000Mb, 10000 Mb/s (16x,1000x)

- Memory Module: 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM (4x,120x)

- Disk : 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)

(latency = simple operation w/o contention  
BW = best-case)



# Results

Distributed Systems show

- high complexity due to a large number of interacting agents
- partial knowledge (about partners, state, time)
- a lot of uncertainty e.g. about crashed nodes

A good introduction: Alvaro Videla, What We Talk About When We Talk About Distributed Systems,

# Liveness and Correctness

<p><b>correctness</b></p> <p>“Something bad will not happen”</p>	<p><b>liveness</b></p> <p>“Something good will eventually happen”</p>
--	---

**The success of a system consists of correctness (expressed by things that should NOT happen) and liveness (expressed as things that SHOULD happen). And all of it is based on failure assumptions (fairness, byzantine errors etc.)**

# **Example: Basic rules of event-based Systems**

## **Correctness:**

- receive notifications only if subscribed to them**
- received notifications must have been published before**
- receive a notification only at most once**

## **Liveness:**

- start receiving notifications some time after a subscription was made**

## **Failure Assumptions:**

- e.g. Fail-stop model with fairness**

**Note that at most once notification may not be enough as it implies that a system crash might lose notifications (better: persistent notifications with exactly once semantics) and also: there is no guarantee about the time lag between subscriptions and the beginning of notifications. No ordering rule is given for a simple system.**

**(see Mühl et.al. 25, 29).**

# Timing Models

- **Synchronous:** Transmit times are strictly defined and events happen at defined moments. Nodes can immediately detect a crashed partner node. Sync. Systems are based on a clock, e.g. CPUs.
- **Asynchronous:** Messages will “eventually” arrive. There is no exact time between sending and receiving messages. Therefore, a node cannot tell, whether another node has crashed or is just very slow to respond. No timeouts exist, because they would require a clock.
- **Partial Synchronous:** asynchronous systems enhanced with local clocks. This is the model that is used for real-world distributed systems.

# Communication Models

- **Message Passing**
- **Shared Memory**

We are going to use the message passing model to build higher abstractions. An important question in this model is about the quality of the connection with respect to losses or duplicated messages. Delivery guarantees like “at-most-once” or “at-least-once” model different failure scenarios.

# FLP: The Impossibility of Consensus in Asynchronous Distributed Systems

The so-called FLP (Fischer, Lynch and Patterson) result proves that any consensus protocol capable of tolerating even a single failure must have non-terminating runs (in which no decision is reached) (Birman)

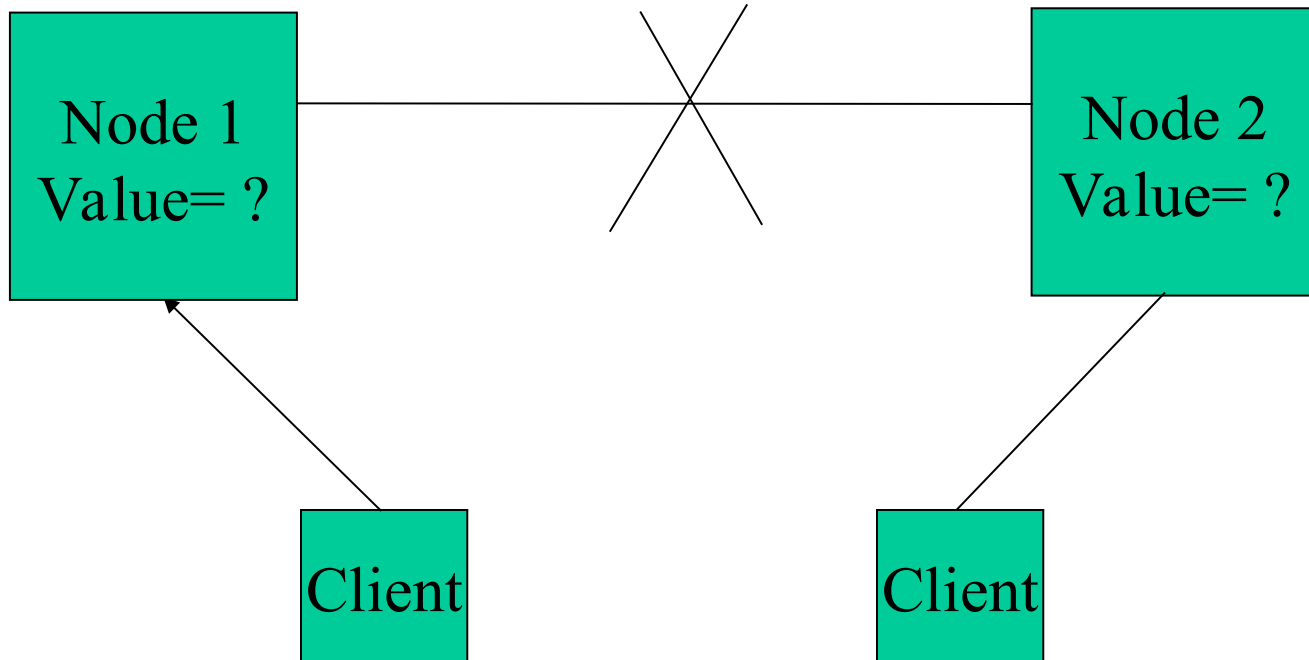
This problem affects basically all consensus based distributed algorithms (like leader election, agreement, replication, locking, atomic broadcast)

The reason lies in the fact, that a unique leader is needed to come up with a decision. But message delays in asynchronous systems can lead to forced re-election of new leaders – delaying the decision forever.

See Ken Birman on FLP (resources),

<http://www.cs.cornell.edu/courses/cs614/2002sp/cs614%20--%20FLP.ppt>

# CAP Theorem



E.Brewer (2000). Chose either Consistency OR Availability in the presence of possible network partitions. Client would either get no answer (consistency) or a possibly incorrect answer (availability)

# CAP Theorem - Preconditions

- Consistency: Atomic, linearizable. Total order on all operations due to linearization point (single instant).
- Availability: Every request received by a non-failing node MUST result in a response. [...] Every request MUST terminate
- Partition Tolerance: The network will be allowed to lose arbitrarily many messages sent from one node to another.

From: Gilbert and Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News (2002) vol. 33 (2) pp. 59. For an explanation of "atomic" see "Atomic Broadcast" below.



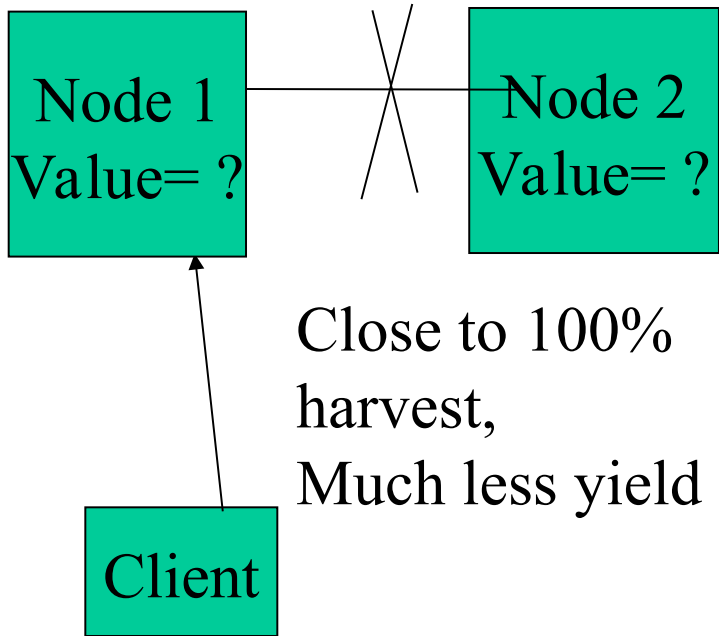
# CAP Theorem – Common Misconceptions

- Consistency: Few systems really achieve a total order of requests! We need to take a closer look at what is possible and at what costs (latency, partial results)
- Availability: Even an ISOLATED NODE with a working quorum (majority) on the other side, needs to answer requests – thereby breaking consistency. The node does not KNOW that a quorum exists...
- Partition Tolerance: You cannot UN-CHOOSE PT. It is always there. CA systems are not possible!

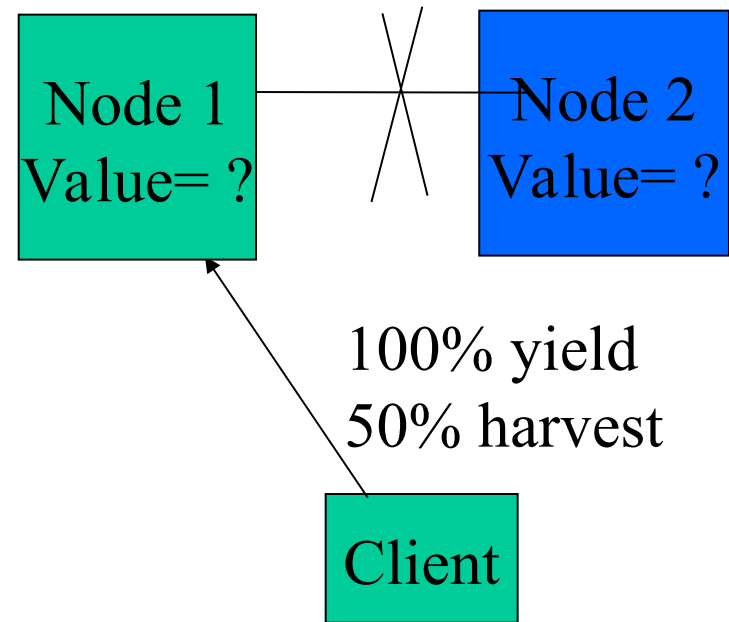
From: Code Hale, You Can't Sacrifice Partition Tolerance and M. Kleppman, Please stop calling databases CP and AP.

# CAP Theorem – Yield and Harvest in AP

Replicated



Sharded



From: Fox and Brewer. Harvest, yield, and scalable tolerant systems. Replication vs. Sharding makes a huge difference in output quality and request numbers.

# CAP Theorem – Modern View

There are more failure types than just PT: Host-crash, client-server disconnect etc. You cannot avoid those failures completely.

Many systems DO NOT NEED linearizability! Chose carefully, what type of consistency you really need.

Most systems chose latency over consistency with availability coming in second.

A fully consistent system in an asynchronous network is impossible (in the sense of FLP). FLP is much stronger than CAP.

Type of architecture (replication, sharding) and client-abilities (failover) have an impact as well.

# From CAP to PACELC?

“because any DDBS must be tolerant of network partitions, according to CAP, the system must choose between high availability and consistency. [...] It is not merely the partition tolerance that necessitates a tradeoff between consistency and availability; rather, it is the combination of

- partition tolerance and
- the existence of a network partition itself.”

## PACELC:

A more complete portrayal of the space of potential consistency tradeoffs for DDBSs can be achieved by rewriting CAP as PACELC (pronounced “pass-elk”): if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?

From: Daniel J. Abadi, Consistency Tradeoffs in Modern Distributed Database System Design, <http://www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf>

# Failures, Types, Models and Detection

# Failures

- Network: partitioning
- CPU/Hardware: instruction failures, RAM failures
- Operating System: Crash, reduced function
- Application: crash, stopped, partially functioning

Unfortunately in most cases there is no failure detection service which would allow others to take appropriate action. Such a service is possible and could detect even partitionings etc. through the use of MIBs, triangulation etc. Applications could track themselves and restart if needed.

# Failure Types

**-Bohr-Bug: shows up consistently and can be reproduced. Easy to recognize and fix.**

**-Heisenbug: shows up intermittently, depending on the order of execution. High degree of non-determinism and context dependency**

**Due to our complex IT environments the Heisenbugs are both more frequent and much harder to solve. They are only symptoms of a deeper problem. Changes to software may make them disappear – for a while. More changes might causes them to show up again. Example: deadlock „solving“ through delays instead of ressource order management.**

# Failure Models

- **Crash-stop: A process crashes atomically and stays down.**
- **Crash-stop with recovery: A process is down from a crash time point to the beginning of its recovery, and up from the beginning of a recovery until the next crash happens. For consensus,  $2f+1$  machines are needed (quorum)**
- **Crash-amnesia: A process crashed and restarts without recollection of previous events/data.**

**Failstop: A machine fails completely AND the failure is reported to other machines reliably.**

- **Omission errors: processes fail to send or receive messages event so they are alive.**

**Byzantine Errors: machines or part of machines, networks, applications fail in unpredictable ways and may recover partially. For consensus, at least  $3f+1$  machines are needed.**

**Many protocols to achieve consistency and availability make certain assumptions about failure models. This is rather obvious with transaction protocols which may assume recovery behavior by its participants if the protocol should terminate. For byzantine error proofs see J. Welch, CSCE 668, DISTRIBUTED ALGORITHMS AND SYSTEMS**



# Failures and Timeouts

**A timeout is NOT a reliable way to detect failure. It can be caused by short interruptions on the network, overload conditions, routing changes etc.**

**A timeout CANNOT distinguish between the different places and kinds of failures.**

**It CANNOT be used in protocols which require failstop behavior of its participants**

**Most distributed systems offer only timeouts for applications to notice problems**

**A timeout does not allow conclusions about the state of participants. It is unable to answer questions about membership (and therefore responsibility). If timeouts ARE used as a failure notification „split-brain“ conditions (e.g. airtraffic control) can result (Birman 248)**

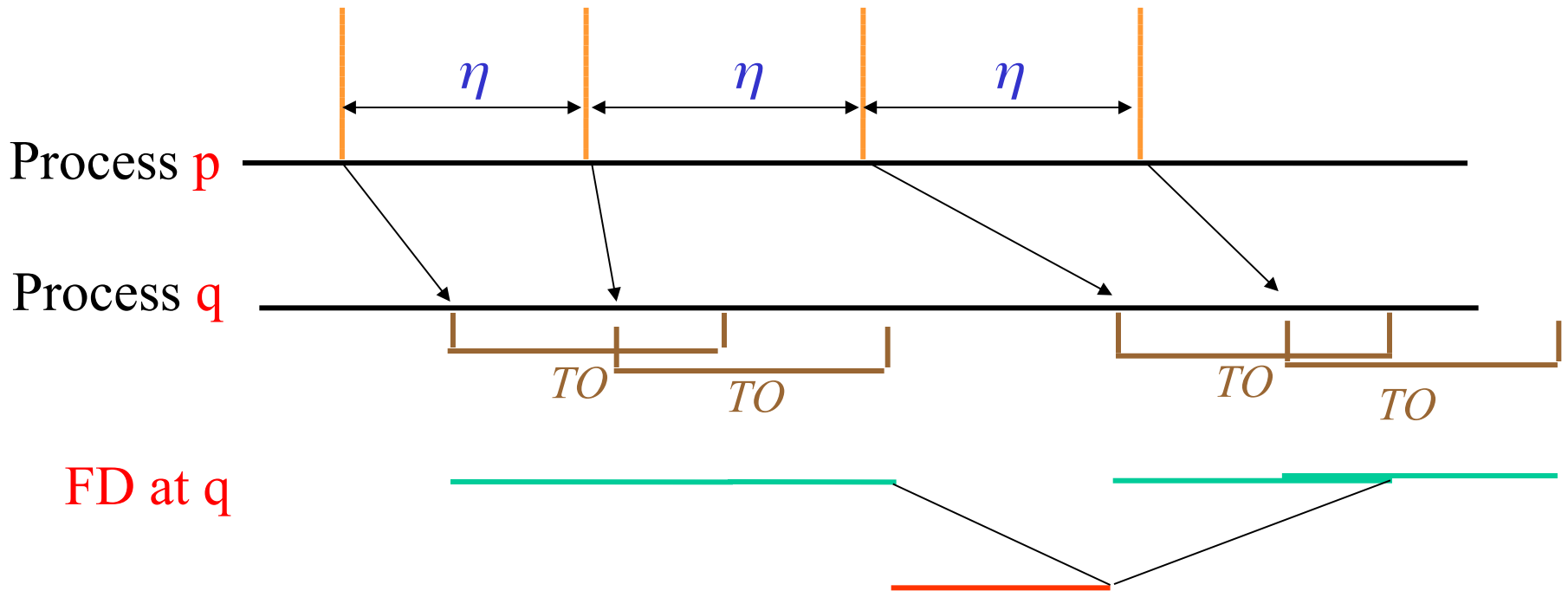
# Failure Detectors

A FD need not be correct all the time. It has to provide the following QoS:

- Be safe all the time, be live during “better” failure periods
- no process blocks forever waiting for a message from a dead coordinator
- eventually some process **x** is not falsely suspected. When **x** becomes the coordinator, every process receives its **x**'s estimate and decides
- do not cause a lot of overhead

(see Sam Toueg, Failure Detectors – A Perspective)

# FD Algorithm (S. Toueg)



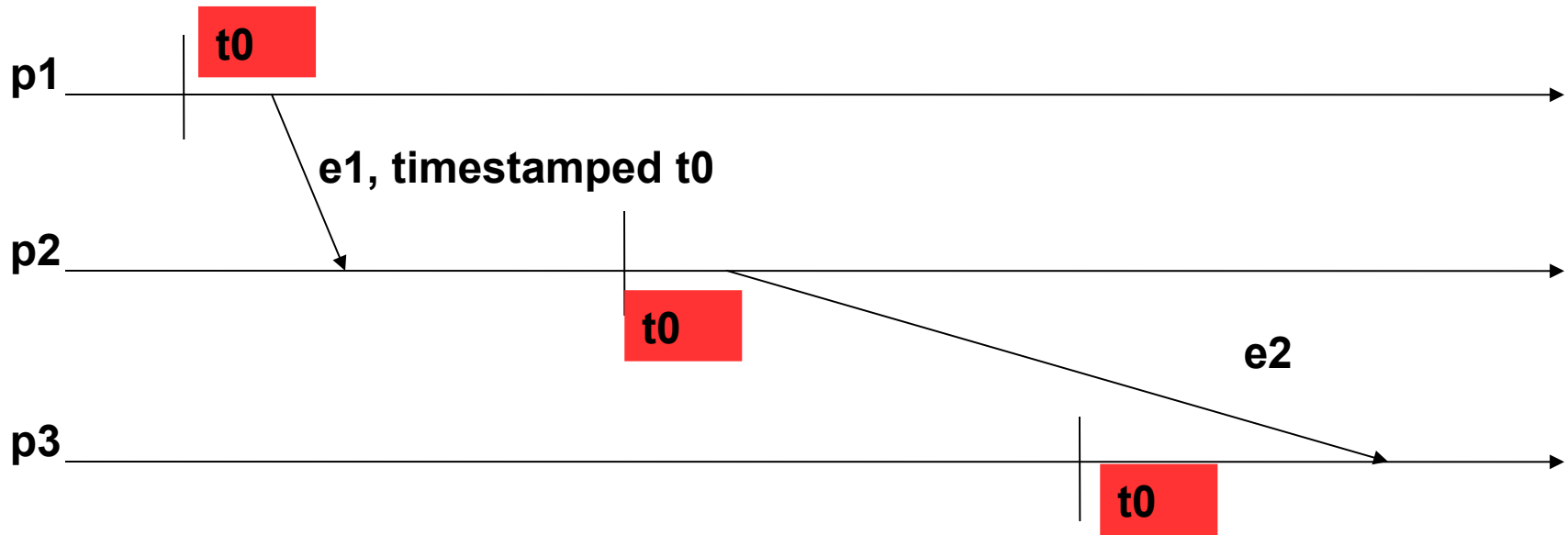
- Timing-out also depends on **previous** heartbeat

# Time in Distributed Systems

- **Event clock time (logical)**
- **vector clock time (logical)**
- **TrueTime (physical interval time)**
- **Augmented time (physical/logical combination)**

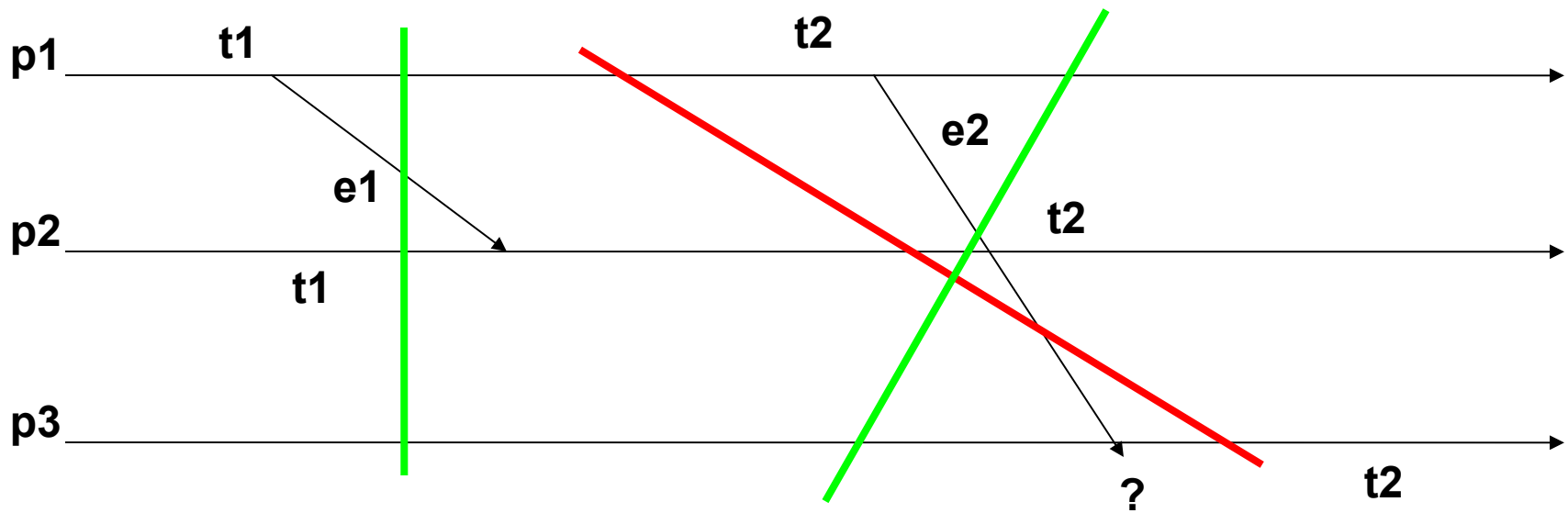
**There is no global time in distributed systems. Logical time modelled as partially ordered events within a process or also between processes. A good read: There is No Now - Problems with simultaneity in distributed systems, Justin Sheehy, ACM Queue. <https://queue.acm.org/detail.cfm?id=2745385>**

# No Global Time in Distributed Systems



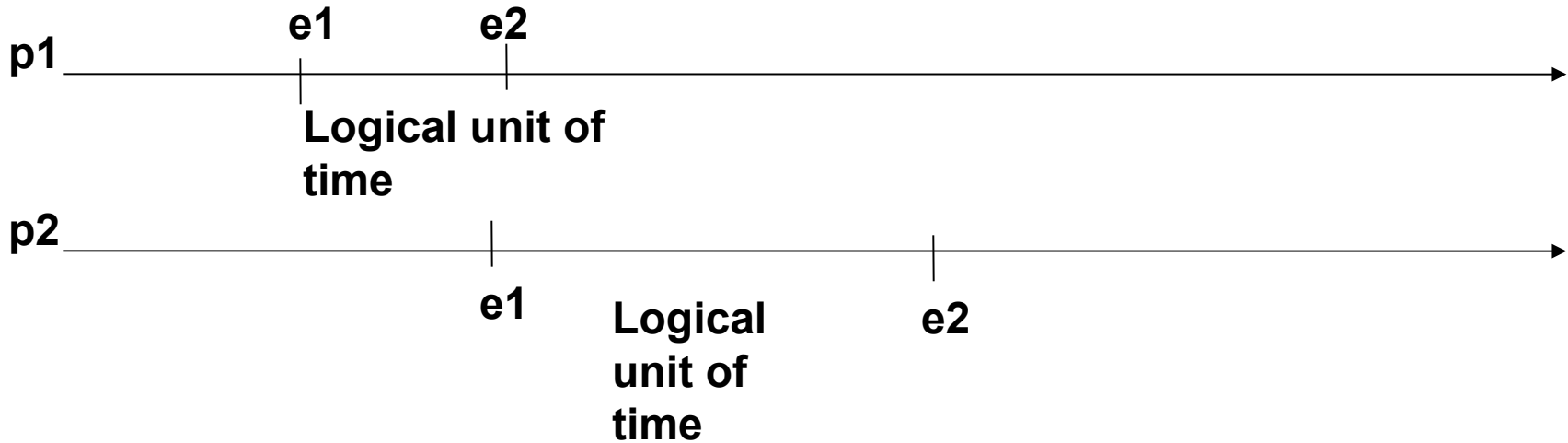
The processes p1-p3 run on different clocks. The clock skew is visible in the distances of  $t_0$  on each time line.  $T_0$  represents a moment in absolute (theoretical) time in this distributed system. For p2 it looks like  $e_1$  is coming from the future (the sender timestamp is bigger than p2's current time).  $E_2$  looks ok for p3. Causal meta-data in the system can order the events properly. Alternatively logical clocks and vector clocks (see Birman) can be used to order events on each process and between processes. This does NOT require a central authority

# Consistent Cuts vs Inconsistent Cuts



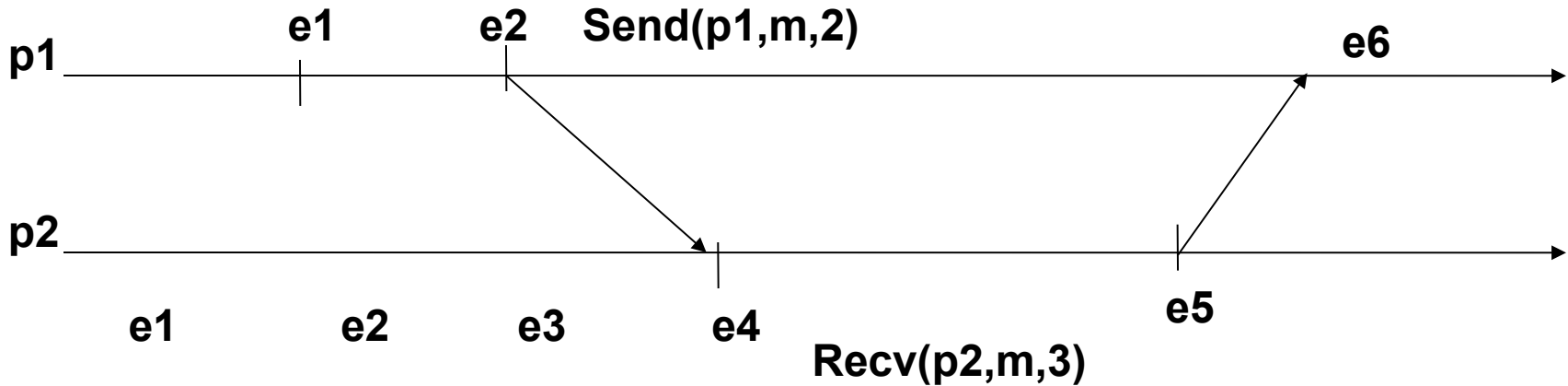
**Consistent cuts produce causally possible events. With inconsistent cuts (red) some events arrive before they have been sent. Consistency of cuts is independent of simultaneous capture. (Birman 257). Tipp: a system with atomic snapshot makes a consistent cut easy by going through the history!**

# Event Clock (Logical Clock)



**Events are partially ordered within processes according to a chosen causal model and granularity.  $E1 < e2$  means e1 happen before e2. The time between events is a logical unit of time. It has no physical extension.**

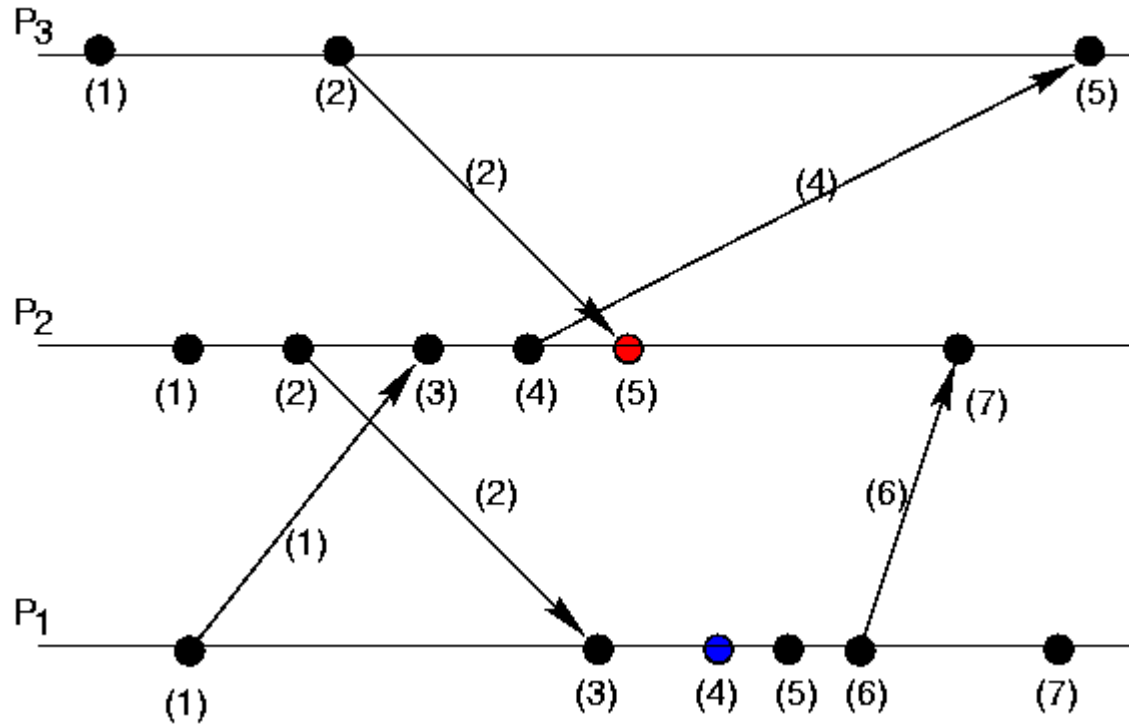
# Event Clock (Logical Clock)



Events delivered through messages clearly relate processes and their times and events. This external order is also a partial order of events between processes:  $\text{send}(p1,m) < \text{recv}(p2,m)$ . The new logical clock value is  $\max(\text{own value} + 1, \text{received value})$ .



# Lamport Logical Clock (2)



# „happens before“ ( $\mathbb{R}$ ) according to Lamport

$A \mathbb{R} B$  if  $A$  and  $B$  are within the same sequential thread of control and  $A$  occurred before  $B$ .

$A \mathbb{R} B$  if  $A$  is the event of sending a message  $M$  in one process and  $B$  is the event of receiving  $M$  by another process. if  $A \mathbb{R} B$  and  $B \mathbb{R} C$  then  $A \mathbb{R} C$ . Event  $A$  *causally affects* event  $B$  iff  $A \mathbb{R} B$ .

Distinct events  $A$  and  $B$  are *concurrent* ( $A \parallel B$ ) if we do not have  $A \mathbb{R} B$  or  $B \mathbb{R} A$ .

**Lamport Logical Clocks** count events and create an ordering relation between them. These counters can be used as timestamps on events. The ordering relation captures all causally related events but unfortunately also many unrelated (concurrent) events thereby creating false dependencies. (Source:

<http://www.cs.fsu.edu/~xyuan/cop5611/lecture5.html> and Ken Birman)

# Vector Clocks

## Vector Clocks

Event  
counter for  
Node i = 2

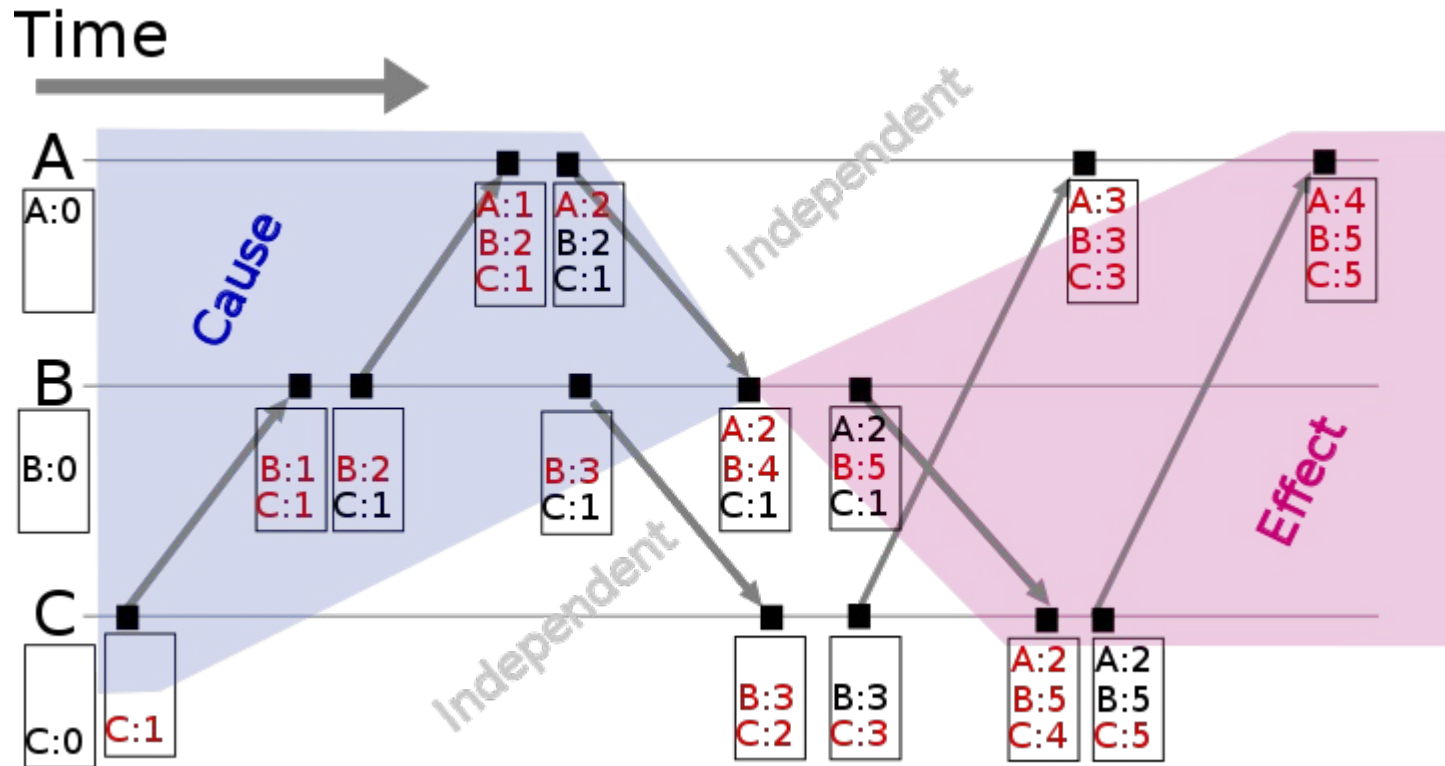
1 | 4 | 6 | 4 | 1 | 4 | 1 | 4 | 1 | 4 | 1 | 4 | 1 | 4 | 2 |

Event  
counter for  
Node j = 4

1 | 4 | 6 | 4 | 7 | 4 | 1 | 4 | 1 | 4 | 1 | 4 | 1 | 4 | 2 |

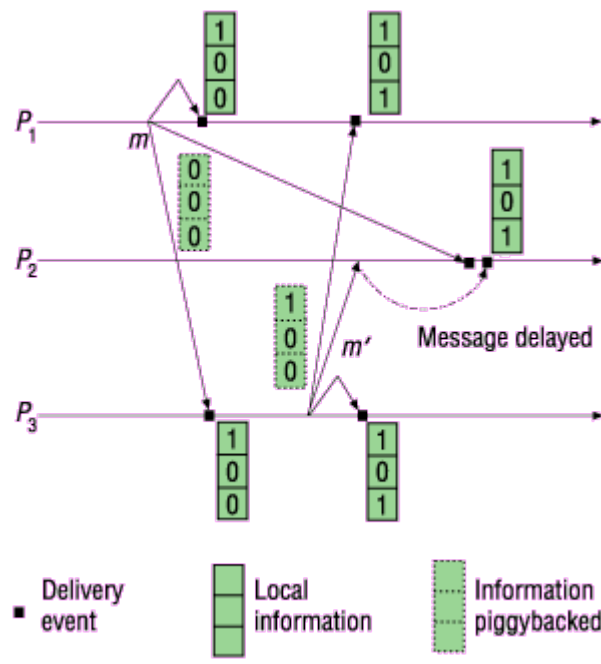
Vector clocks are transmitted with messages and compared at the receiving end. If for all positions in two vector clocks A and B the values in A are larger than or the same as the values from B we say that Vector Clock A dominates B. This can be interpreted as potential causality to detect conflicts, as missed messages to order propagation etc.

# Causal dependencies with vector clocks



Source: wikipedia commons

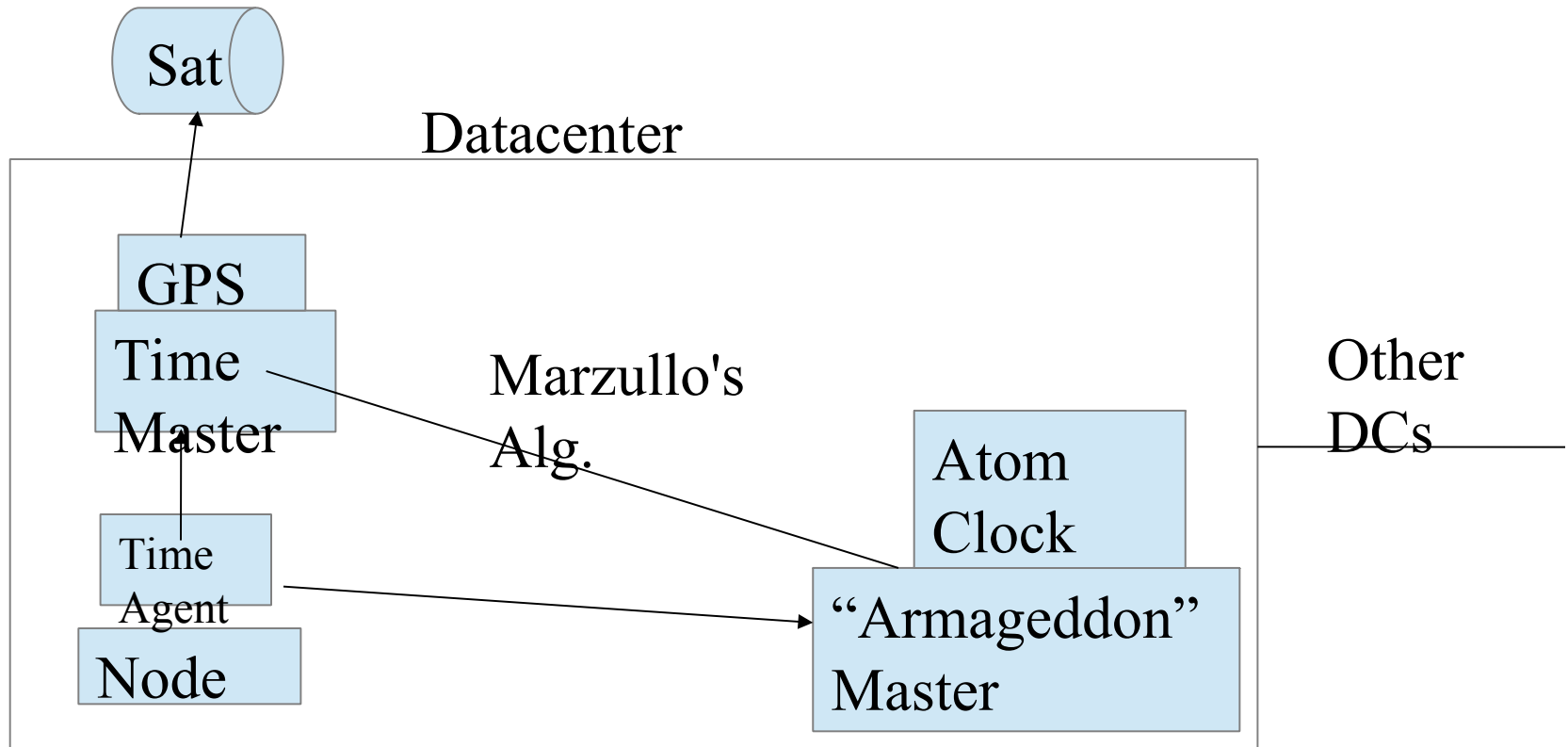
# Causal Re-ordering of messages



From: Roberto Baldoni et.al,

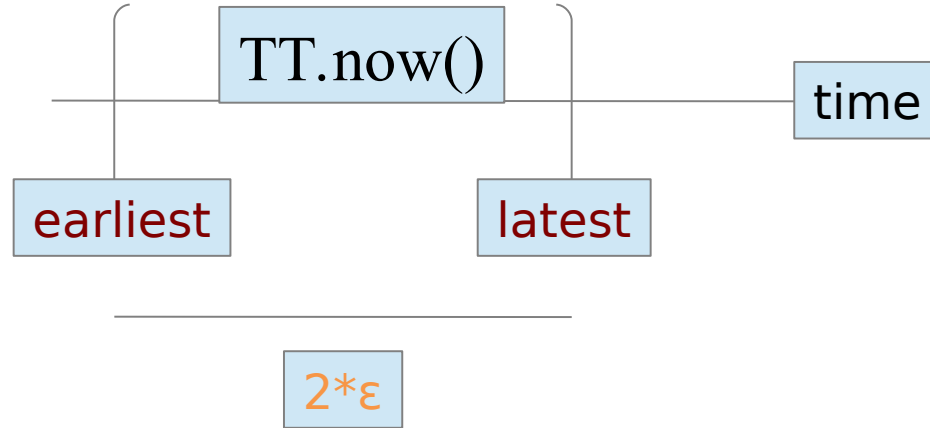
[http://net.pku.edu.cn/~course/cs501/2008/reading/a7\\_tour\\_vc.html](http://net.pku.edu.cn/~course/cs501/2008/reading/a7_tour_vc.html)

# Physical Interval Time (e.g. TrueTime)



Communicating time masters ensure, that there is a bounded time-uncertainty for the event time  $e$  with respect to the absolute (wall-clock) time. Time servers check themselves for rogue clocks. Error is in the area of 6ms. 38

# Interval Time



The system guarantees, that for an invocation  $tt = TT.now()$ ,  $tt.earliest() \leq tabs(enow) \leq tt.latest()$ , where `enow` is the invocation event. (from: Spanner: Google's Globally-Distributed Database, James Corbett et.al.). Does Spanner beat CAP? Interesting paper from E.Brewer, Spanner, TrueTime & The CAP Theorem, <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45855.pdf>

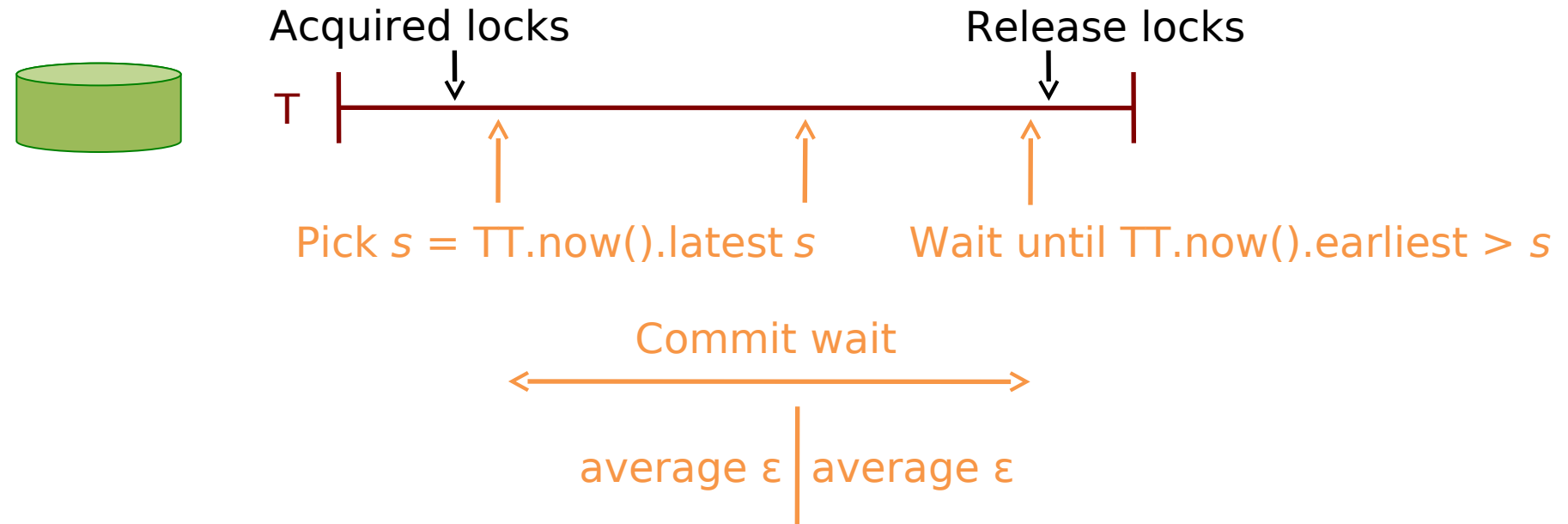
# TrueTime API

TT.now()	TTinterval: [earliest, latest]
TT.after(t)	True if t has passed
TT.before(t)	True if t has not arrived yet
2e	T(now

Using TT, Spanner assigns globally-meaningful commit timestamps to transactions respecting the serialization order: if a transaction T1 commits (in absolute time) before another transaction T2 starts, then T1's assigned commit timestamp is smaller than T2's.



# Commit Time



We will look at distributed transactions and consistency in our session on services! From: OSDI 2012, Google

# Reasons for Hybrid Clocks

- In a large distributed system (e.g. one spanning several data-centers across the world), vector clocks become too large to maintain efficiently.
- Physical interval time needs to respect the uncertainty bound. This forces writes and reads to wait until the interval time is over on all machines.

# Hybrid Logical Clock Algorithm

*Initially*  $l.j := 0; c.j := 0$

## **Send or local event**

$l'.j := l.j;$

$l.j := \max(l'.j, pt.j);$

If  $(l.j = l'.j)$  then  $c.j := c.j + 1$

Else  $c.j := 0;$

Timestamp with  $l.j, c.j$

## **Receive event of message $m$**

$l'.j := l.j;$

$l.j := \max(l'.j, l.m, pt.j);$

If  $(l.j = l'.j = l.m)$  then  $c.j := \max(c.j, c.m) + 1$

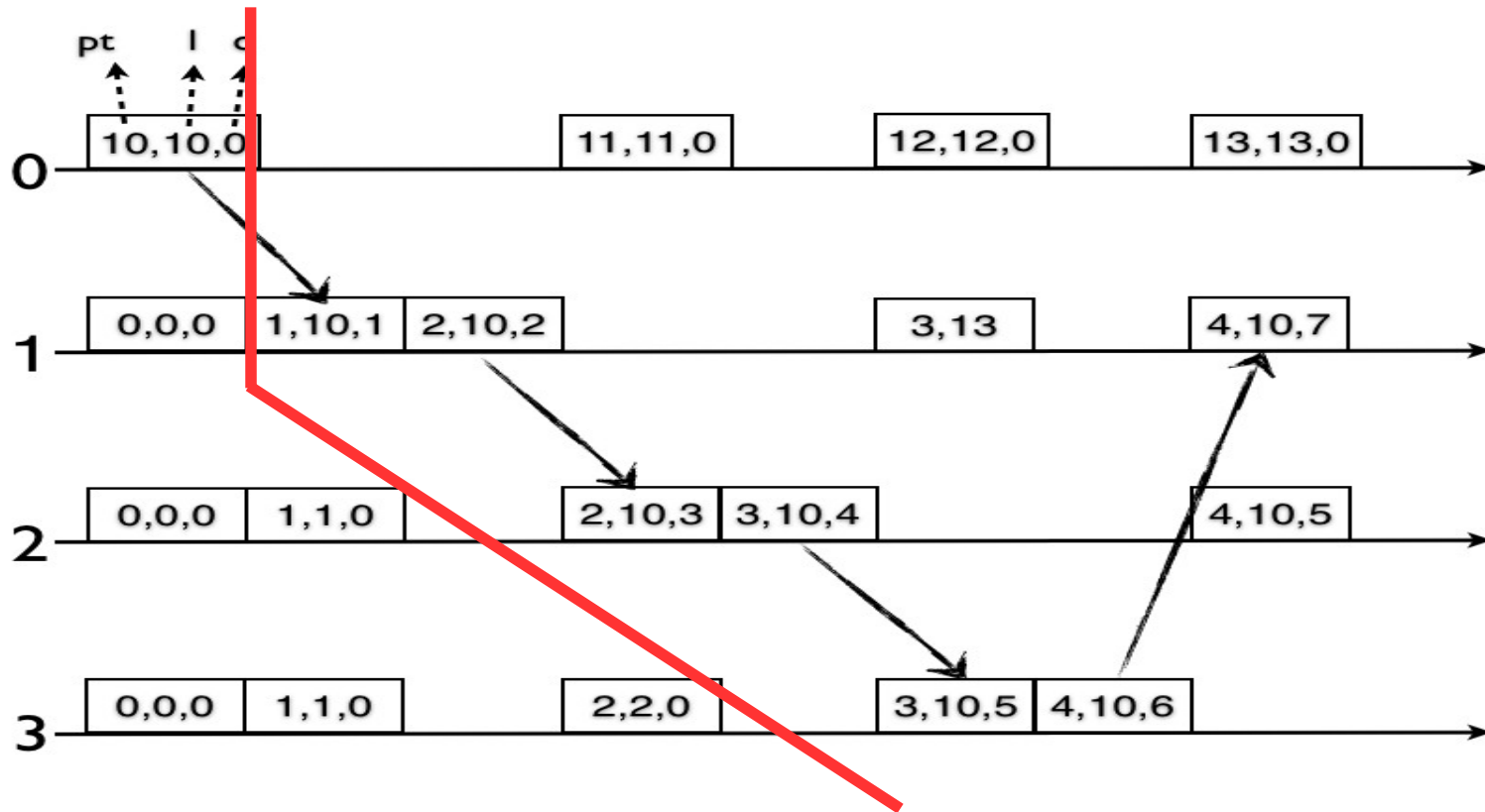
Elseif  $(l.j = l'.j)$  then  $c.j := c.j + 1$

Elseif  $(l.j = l.m)$  then  $c.j := c.m + 1$

Else  $c.j := 0$

Timestamp with  $l.j, c.j$

# Hybrid Logical Clock Run



<http://www.cse.buffalo.edu/tech-reports/2014-04.pdf>

Notice a consistent cut at  $l=10$  and  $c=0$

# Ordering, Causality and Consensus

At the end of the day all coordination in distributed systems is based on ordering events!

# Ordering in Distributed Event-Systems

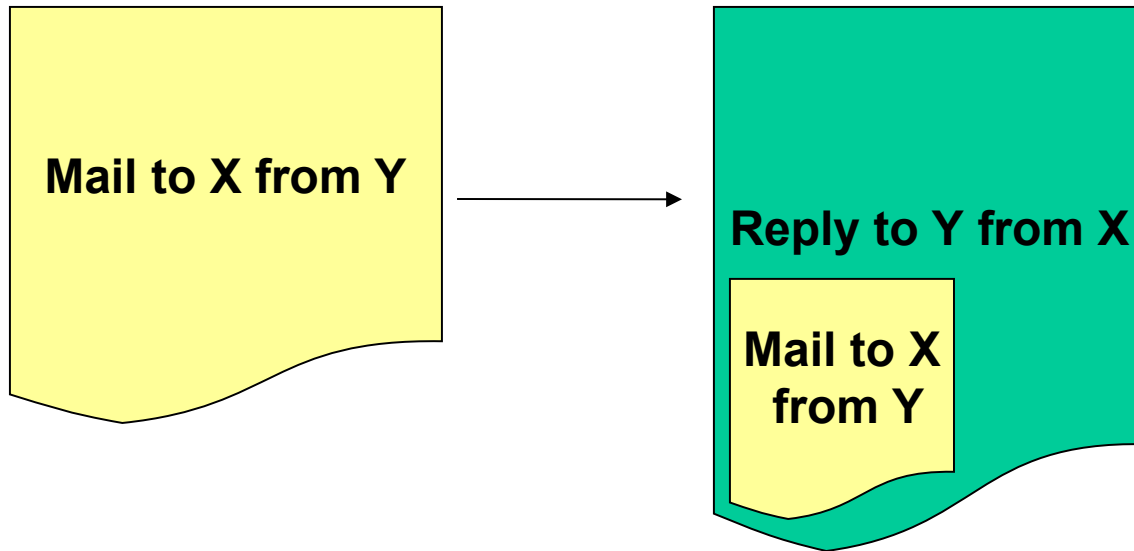
**-FIFO Ordering: a component needs to receive notifications in the order they were published by the publisher**

**-Causal Ordering**

**-Total Ordering: events  $n_1$  and  $n_2$  are published in this order. Once a component receives  $n_2$  not other component in this system is allowed to receive  $n_1$ . (Solution: one component decides about the global order of events)**

**Total Ordering is orthogonal to the other two ordering relations. The most important question is when a component is allowed to accept a message and publish it.**

# Causality Levels?



**Does a reply to a mail imply a causal relationship between both events? What if the reply comes long after the original mail and the author only used the reply-feature for convenience reasons (just re-use the senders mail address as a new target instead of typing the address). The original mail content would not have a connection to the reply content. But would the reply mail have happened without the original mail? Perhaps. But is there a computing relation between both? Perhaps.**

# Consensus

A set of processes have inputs  $v_i \in \{0,1\}$

A Protocol is started (by some sort of trigger)

Objective: all decide  $v$ , for some  $v$  in the input set

Example solution: “vote” and take the majority value  
(After Birman)

Termination: Every correct process eventually decides some value.

Validity: If a process decides  $v$ , then  $v$  was proposed by some process.

Integrity: No process decides twice.

Agreement: No two correct processes decide differently.

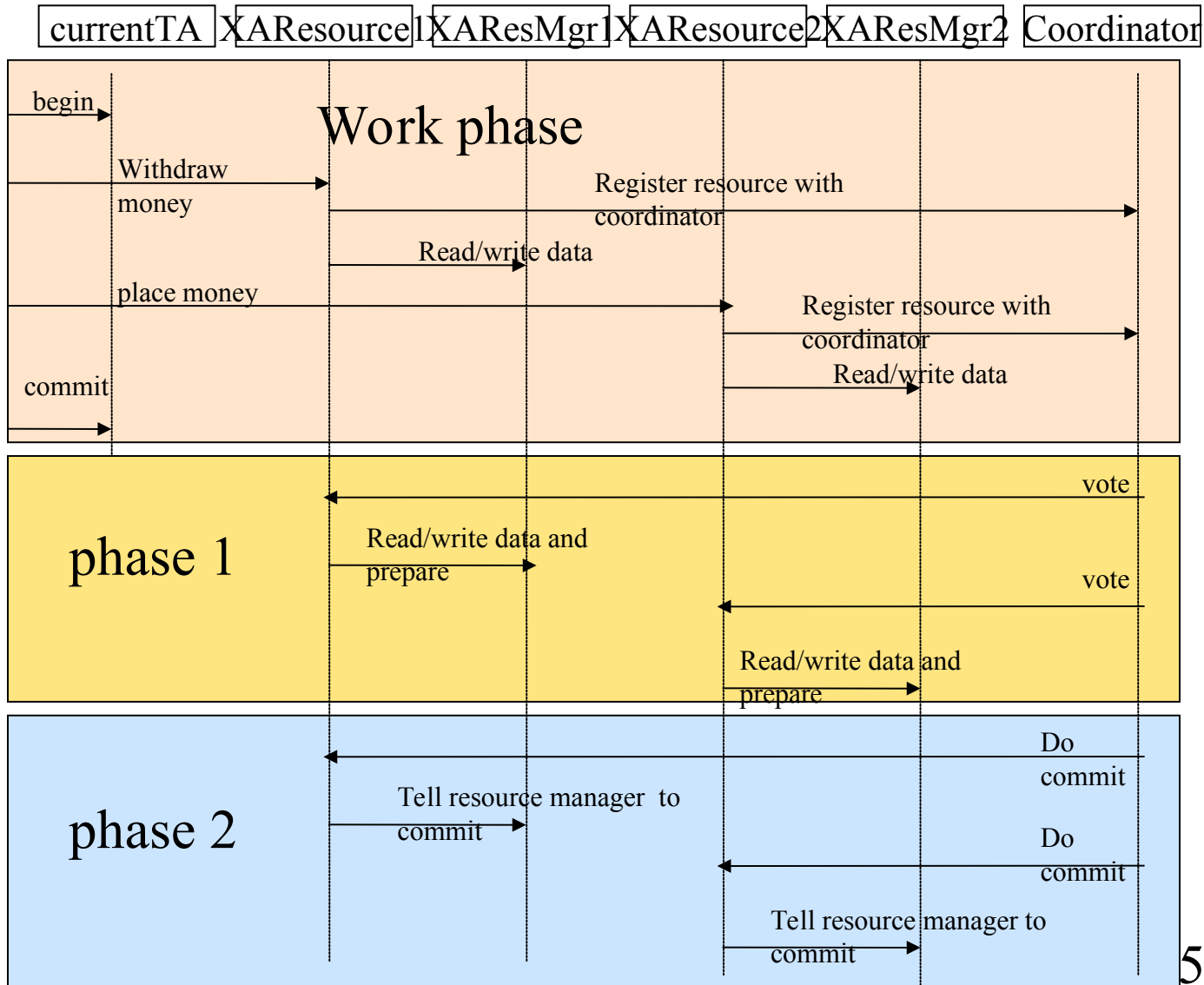


# Algorithms for Consensus

- Two-Phase Commit
- Static Membership Quorum, Paxos, Raft
- Dynamic Group Membership (virtual synchrony, multicast based)
- Gossip Protocols

These protocols offer trade-offs with respect to correctness, liveness (availability, progress) and performance

# Two-Phase Commit (2PC)



Participants can abort TA anytime

After 1<sup>st</sup> OK Participants cannot abort/make progress without coordinator

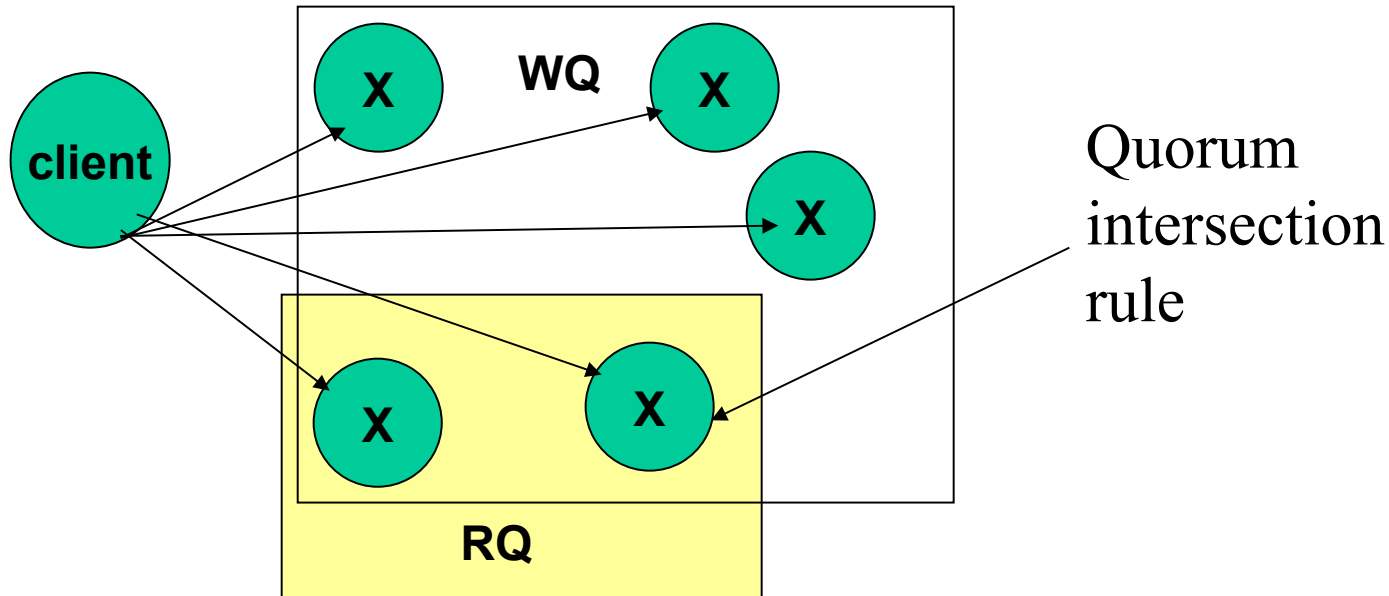
No progress without coordinator

# Liveness and Correctness of 2PC

- 2PC allows atomic (linearizable) updates to participants. But this depends on the Type of implementation (e.g. read locks)
- 2PC is considered rather expensive with respect to latency. Because participants are Not allowed to step back from a decision in the prepare and commit phases, a persistent log is required at each participant.
- The crash-failure model can bring the execution of 2PC to a halt, e.g. when the Coordinator fails and stays down.
- The crash-failure of a participant after the work-phase leads to a heuristic outcome, if it cannot reboot.
- If everything goes right, 2PC has an easy and clear semantic for application developers.

If we assume a different failure model (e.g. fail-stop with detection), we could improve the protocol e.g. by letting participants ask each other about the outcome or deciding on a new leader (coordinator).

# Quorum based Consensus



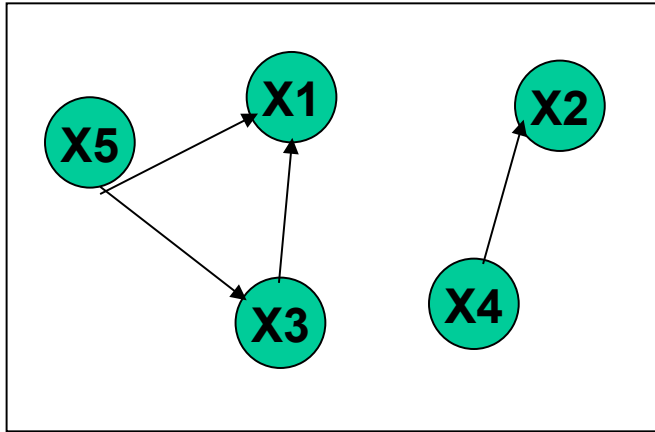
**Majority rulez! Not all known processes  $N$  that replicate a value  $X$  need to be reached in every update. But the number of processes to read (Read Quorum  $RQ$ ) and the number to write (Write Quorum  $WQ$ ) need to be at least  $N+1$ . E.g.  $RQ==2$  and  $WQ==4$  in a system of  $N==5$ . Will partitions affect correctness or liveness? How about performance of a quorum system?**

# Liveness and Correctness of Quorum Protocols

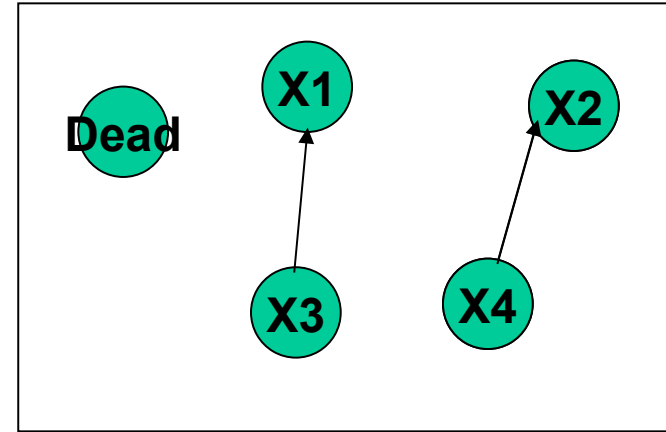
- QP allow atomic (linearizable) updates to participants. But this depends on the Type of implementation (e.g. read quora)
- QP are considered rather expensive with respect to latency. For consistency, even Simple reads need to ask for a quorum. Therefor many QP systems chose a leader Approach to achieve better performance. Routing client requests through a leader raises the problem of leader crashes. Those are usually handled through leases, Which unfortunately cause waits.
- The crash-failure model does not stop the protocol, as long as a quorum is still possible
- Network partitions either force the system to answer with all non-failing nodes (giving up consistency) or to stop answering requests from minority nodes (giving Up availability).

Many of today's scheduling and locking components (chubby, zookeeper, copy-cat etc.) rely on quorum decisions.

# Static Membership and Split-Brain Condition



Majority OK

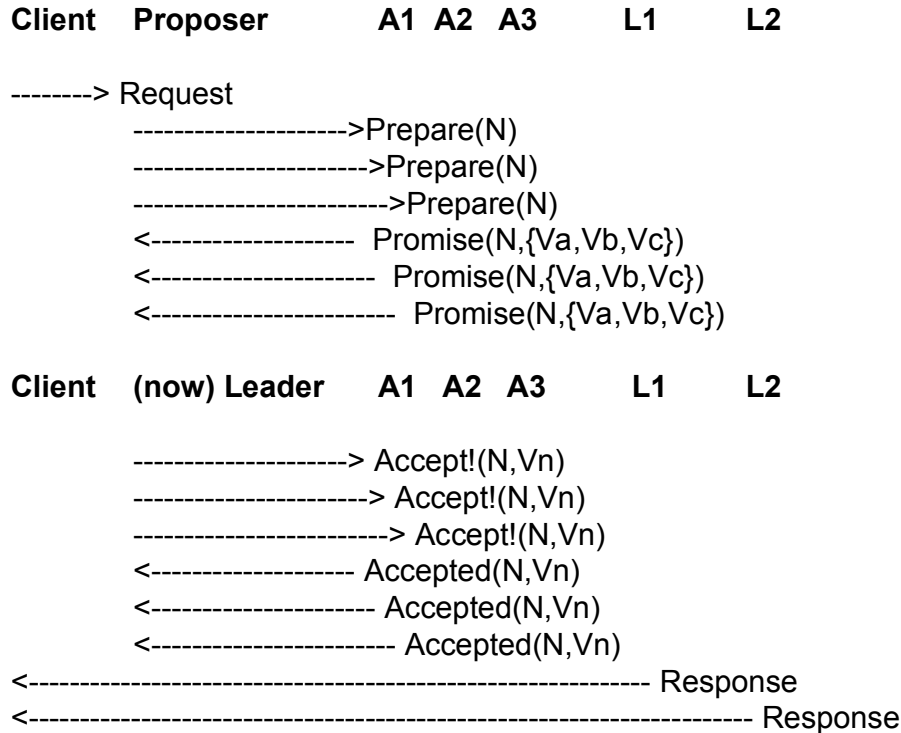


No Majority, X4  
now highest IP

**What is a good group size for quorum algorithms? How can split brain conditions be detected or avoided? How should X4 behave? In a consistent system (CP), the minority cannot know, if it is the only survivor!**

# Quorum based sync. consensus: Paxos

[Message Flow : Basic Paxos (one instance, one successful round)]



(modified after [Turner], A= Acceptor, L=Learner, N=Instance, V=Value)

How does a rather static group leader affect correctness and liveness?  
 (see google chubby paper)? Good talk by Ousterhout on Paxos.

# Paxos Phases

## **Phase 1a: Prepare**

A Proposer (the leader) selects a proposal number N and sends a Prepare message to a Quorum of Acceptors.

## **Phase 1b: Promise**

If the proposal number N is larger than any previous proposal, then each Acceptor promises not to accept proposals less than N, and sends the value it last accepted for this instance to the Proposer (the leader).

Otherwise a denial is sent.

## **Phase 2a: Accept!**

If the Proposer receives responses from a Quorum of Acceptors, it may now Choose a value to be agreed upon. If any of the Acceptors have already accepted a value, the leader must Choose a value from this set. Otherwise, the Proposer is free to propose any value.

The Proposer sends an Accept! message to a Quorum of Acceptors with the Chosen value.

## **Phase 2b: Accepted**

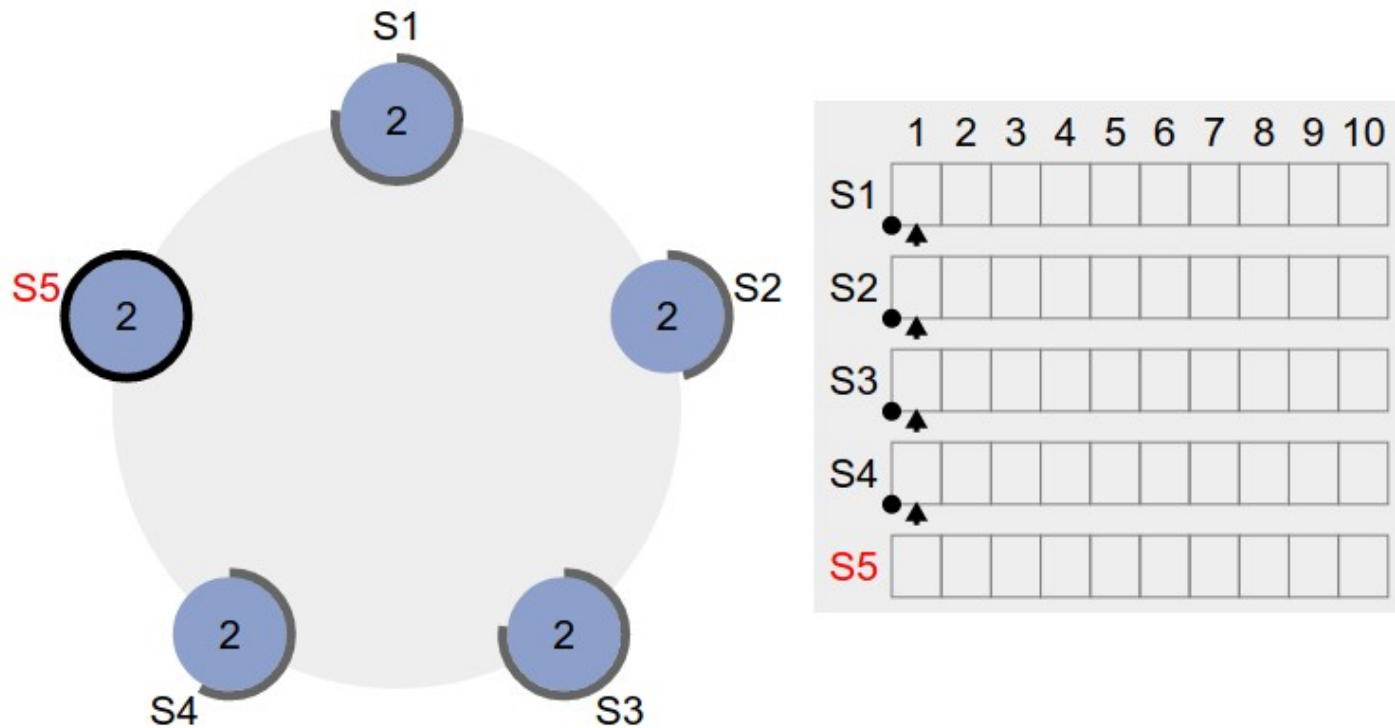
If the Acceptor receives an Accept! message for a proposal it has promised, then it Accepts the value.

Each Acceptor sends an Accepted message to the Proposer and every Learner.

After: The Paxos Family of Consensus Protocols  
Bryan Turner, and L.Lamport, Paxos mad Simple

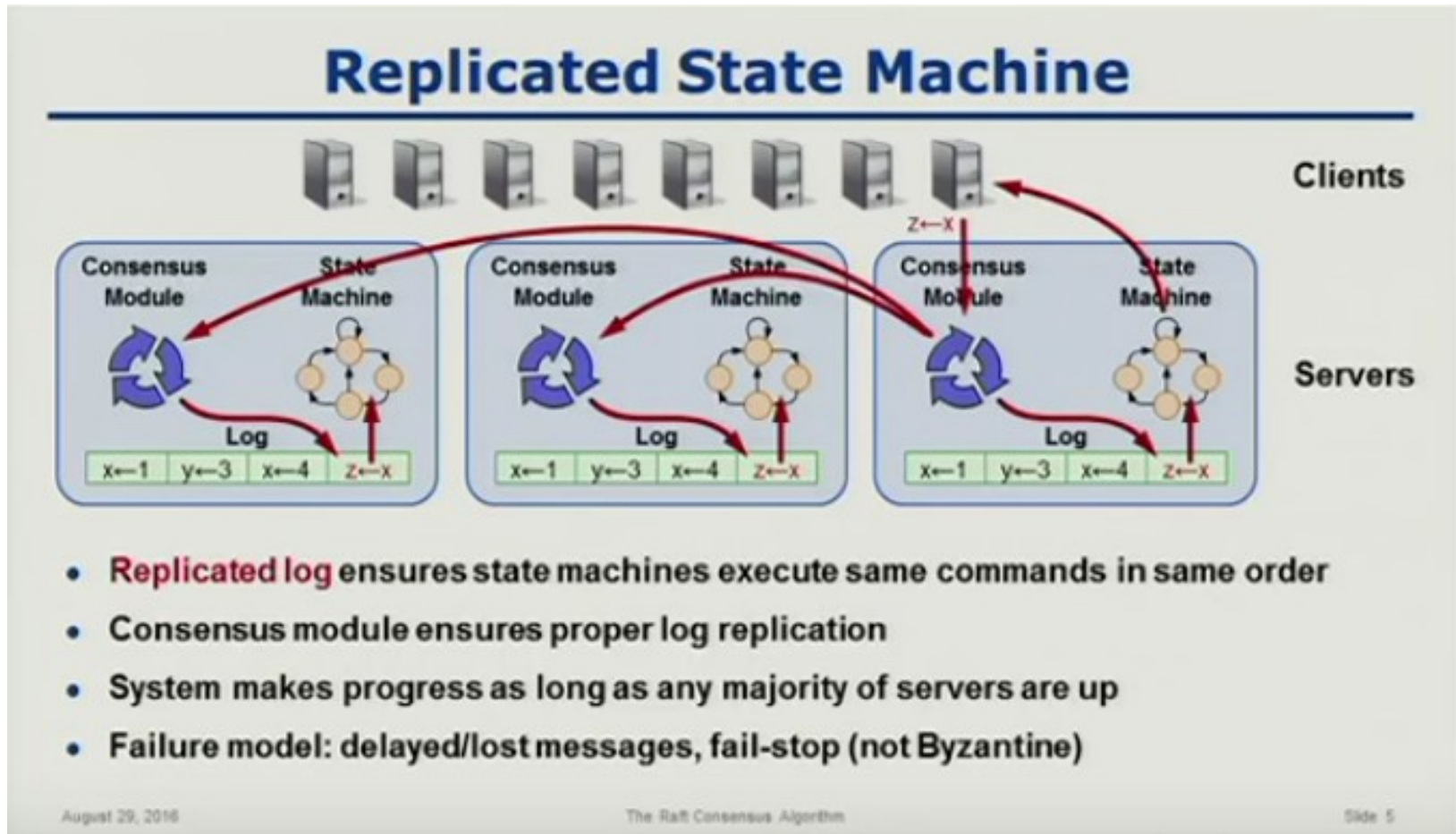


# Understandable Consensus: RAFT



<http://thesecretlivesofdata.com/raft/> and  
<https://raft.github.io/>

# From Consensus to RSM



From: D.Ongaro, Designing for Understandability: The Raft Consensus Algorithm,  
<https://www.youtube.com/watch?v=vYp4LYbnnW8>

# Atomic Broadcast

Validity: If a correct process broadcasts a message, then all correct processes will eventually deliver it.

Uniform Agreement: If a process delivers a message, then all correct processes eventually deliver that message.

Uniform Integrity: For any message  $m$ , every process delivers  $m$  at most once, and only if  $m$  was previously broadcast by the sender of  $m$ .

Uniform Total Order: If processes  $p$  and  $q$  both deliver messages  $m$  and  $m_0$ , then  $p$  delivers  $m$  before  $m_0$  if and only if  $q$  delivers  $m$  before  $m_0$ .

A broadcast algorithm transmits a message from one process (the primary process) to all other processes in a network or in a broadcast domain, including the primary. Atomic broadcast protocols are distributed algorithms guaranteed either to correctly broadcast or to abort without side effects. It is a primitive widely used in distributed computing for group communication. Atomic broadcast can also be defined as a reliable broadcast that satisfies total order (ZooKeeper's atomic broadcast protocol: Theory and practice, Andre Medeiros, March 20, 2012. See also: Xavier Defago, Andre Schiper, and Peter Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv., 36(4):372-421, 2004.)

# Atomic Broadcast Protocol Data

Tuple (e, v, tc) with

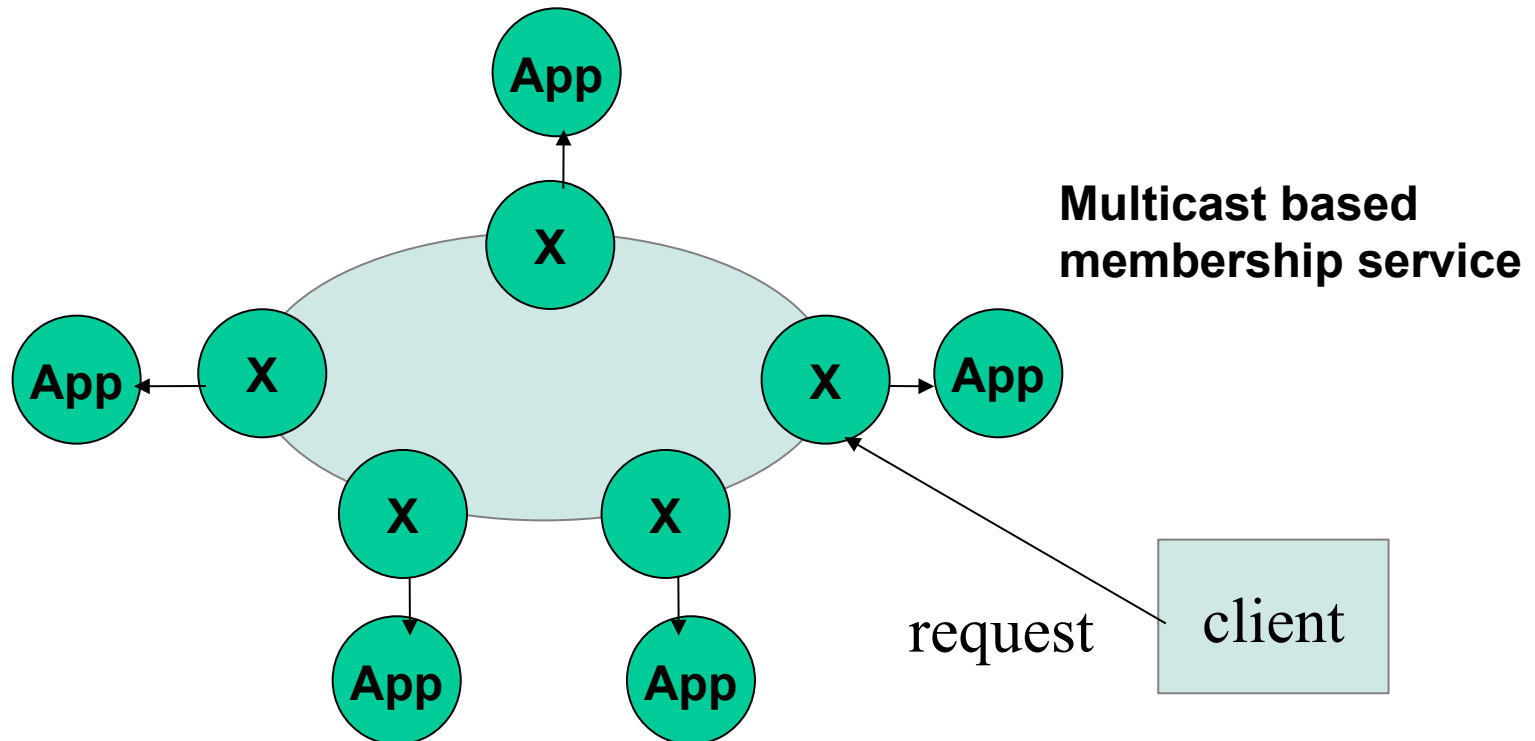
- e being the epoch (the duration of a specific leadership)
- v being the view (defined membership set which lasts until an existing member leaves or comes back?)
- tc being the transaction counter, counting rounds of executions, e.g. updates to replicas

# Atomic Broadcast Protocol Phases

1. Leader election/discovery (members decide on a new leader, a consistent view of the group is formed)
2. Synchronization/recovery (Leader gathers outstanding, uncommitted requests recorded at members, e.g. due to the previous leader crashing. Members missing certain data are updated, until all share the same state)
3. Working (Leader proposes new transactions to the group, collects confirmations and sends out commits.)

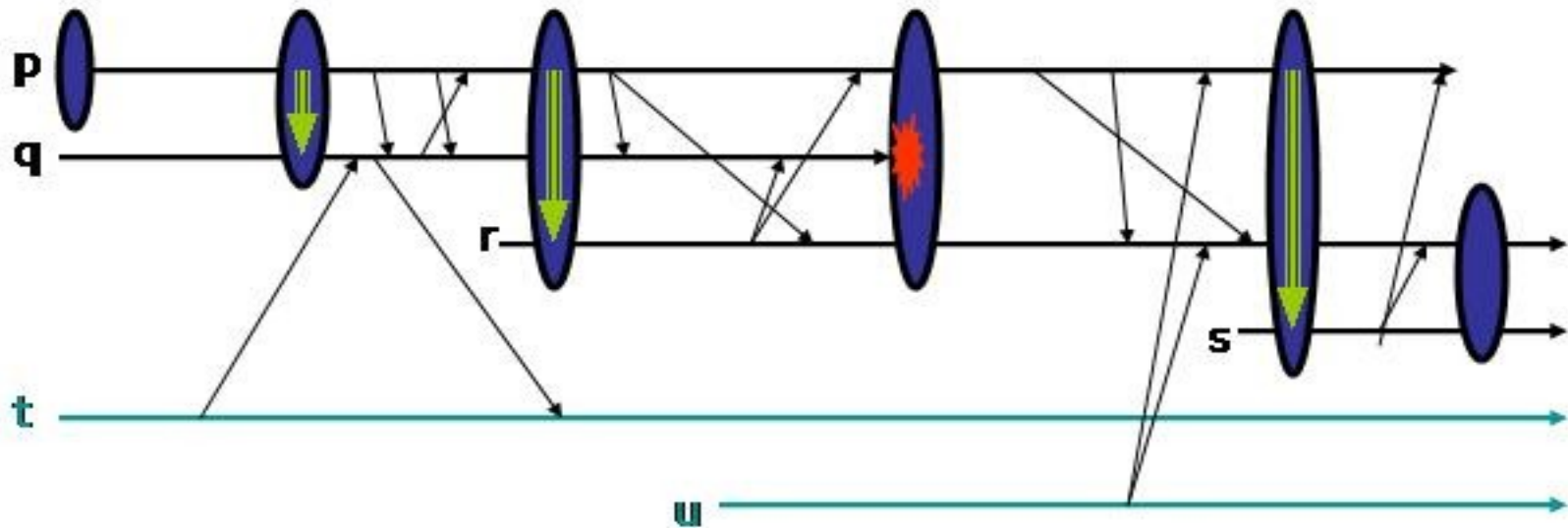
The exact procedures differ from protocol to protocol (Paxos, Raft, etcd). Re-ordering of requests is sometimes critical (e.g. not allowed in zookeeper). Frequent leader change is almost a case of DOS due to the overhead involved. Watch out for latency on the leader node!

# Dynamic Group Membership Services



**Examples are: Horus, Spread. (Birman 251ff). Scales almost linearly up to 32 members. 80000 updates in a group of five (no disk access, failstop nodes). Group members typically do not wait for acknowledge. Models with different ordering and causality qualities exist (fbcast, cbcast, abcast). See Ken Birman's text on wikipedia.**

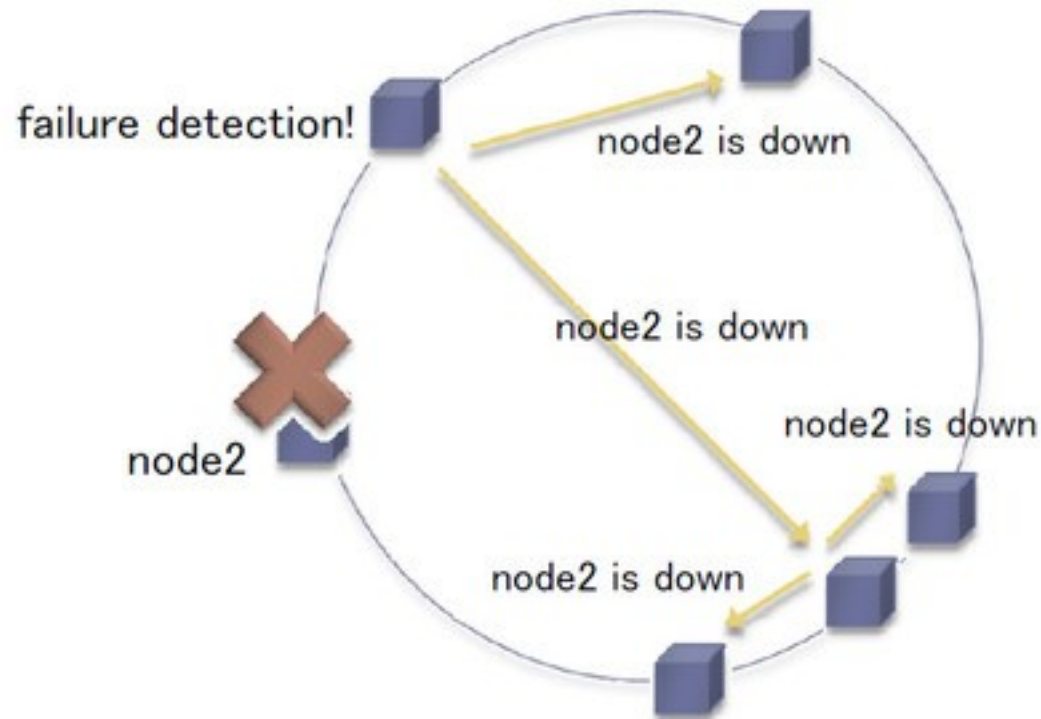
# Virtual Synchrony at a glance



- **We use the weakest (hence fastest) form of communication possible**

From Ken Birman, <https://www.cs.cornell.edu/ken/History.pdf> For the application the events look synchronous. VS alone does not solve consensus!

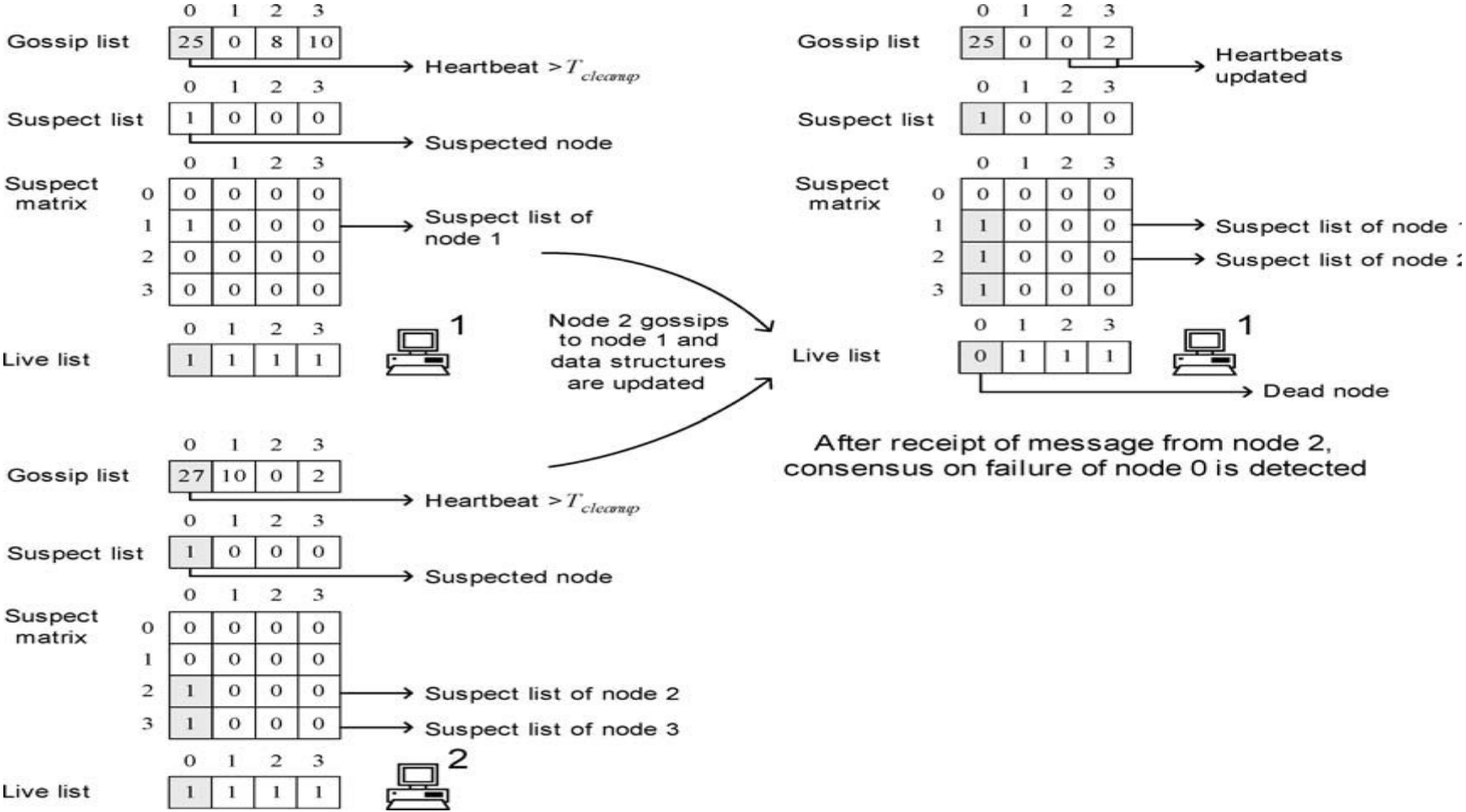
# Gossip Protocols for Consensus



<http://sourceforge.net/apps/mediawiki/kai/index.php?title=Introduction>  
. Layered gossip protocols can provide more efficient communication.



# Example: Monitoring with Gossip Prot. (GEMS)

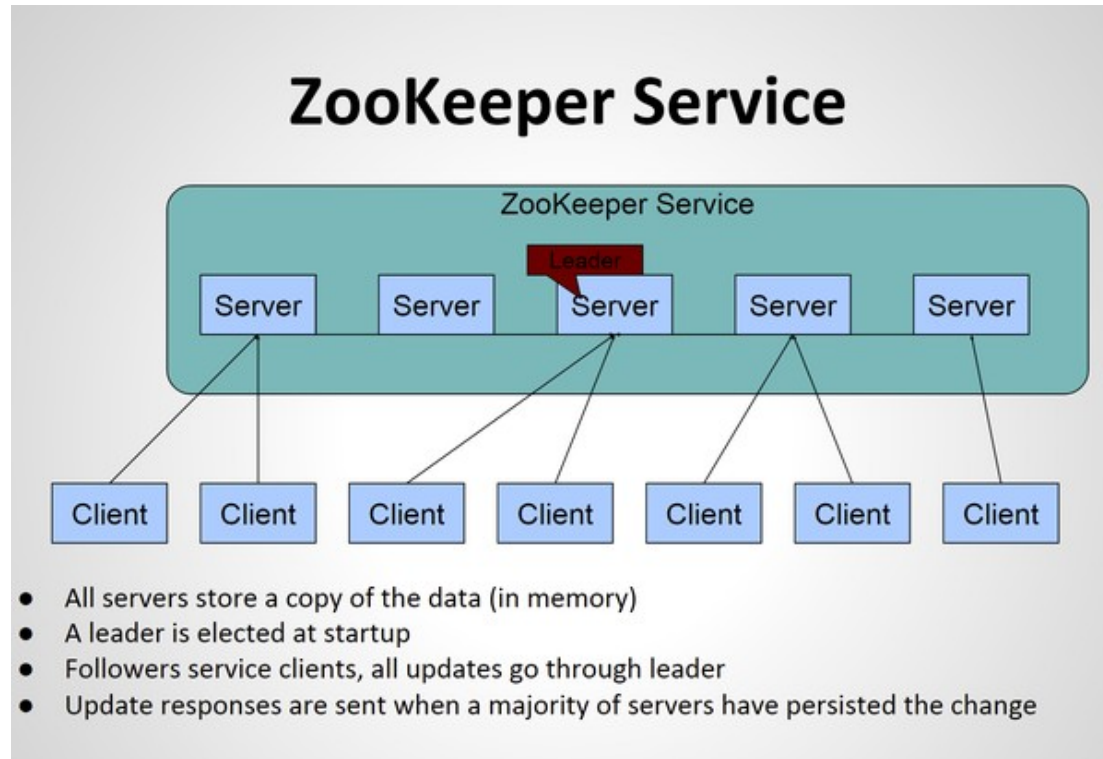


Critical parameters:  $T_{gossip}$ ,  $T_{cleanup}$ ,  $T_{consensus}$ , event propagation, Node selection etc., Rajagopal Subramaniyan et.al., 65

# Synchronous Replication – Distributed Write-Ahead-Log (DWAL)

(CP of CAP)

# Topology of a DWAL

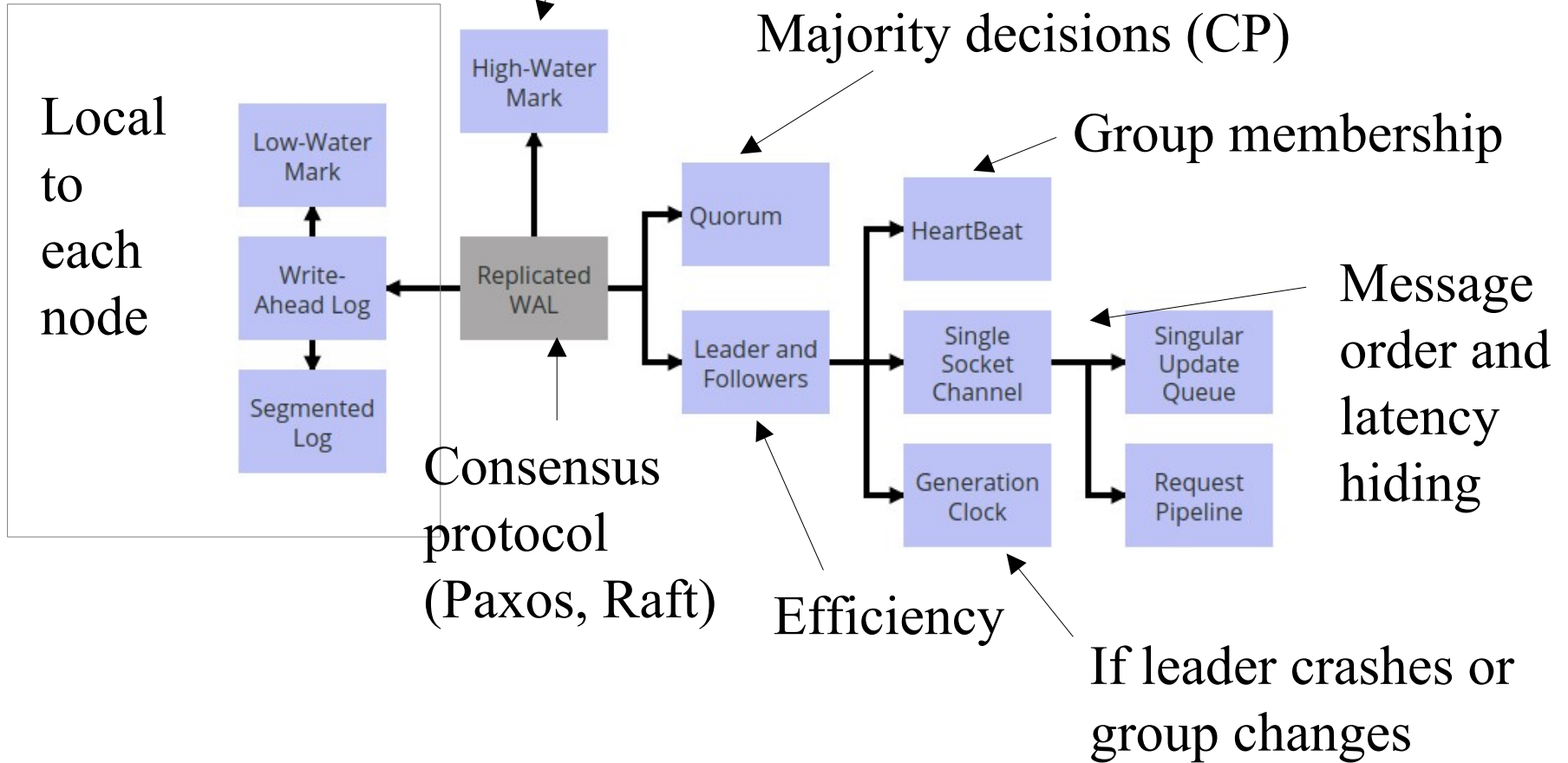


Source:

<https://cwiki.apache.org/confluence/display/ZOOKEEPER/ZooKeeperPresentations?preview=/4193445/91554201/Apache%20ZooKeeper%20-%20Mesosphere.pdf>

# Design Componentes of a Replicated DWAL

Global visibility of replicated state



# Concept Exercise: Correctness and Liveness

- If no quorum can be reached ( $n-2-1$  machines dead/unreachable)?
- if a leader crashes?
- if a follower crashes?
- How available is a DWAL? (scenarios)
- The distributed state machine approach: availability?
- Is the DWAL coherent and consistent?

# Consistency in Non-Transactional Distributed Storage Systems

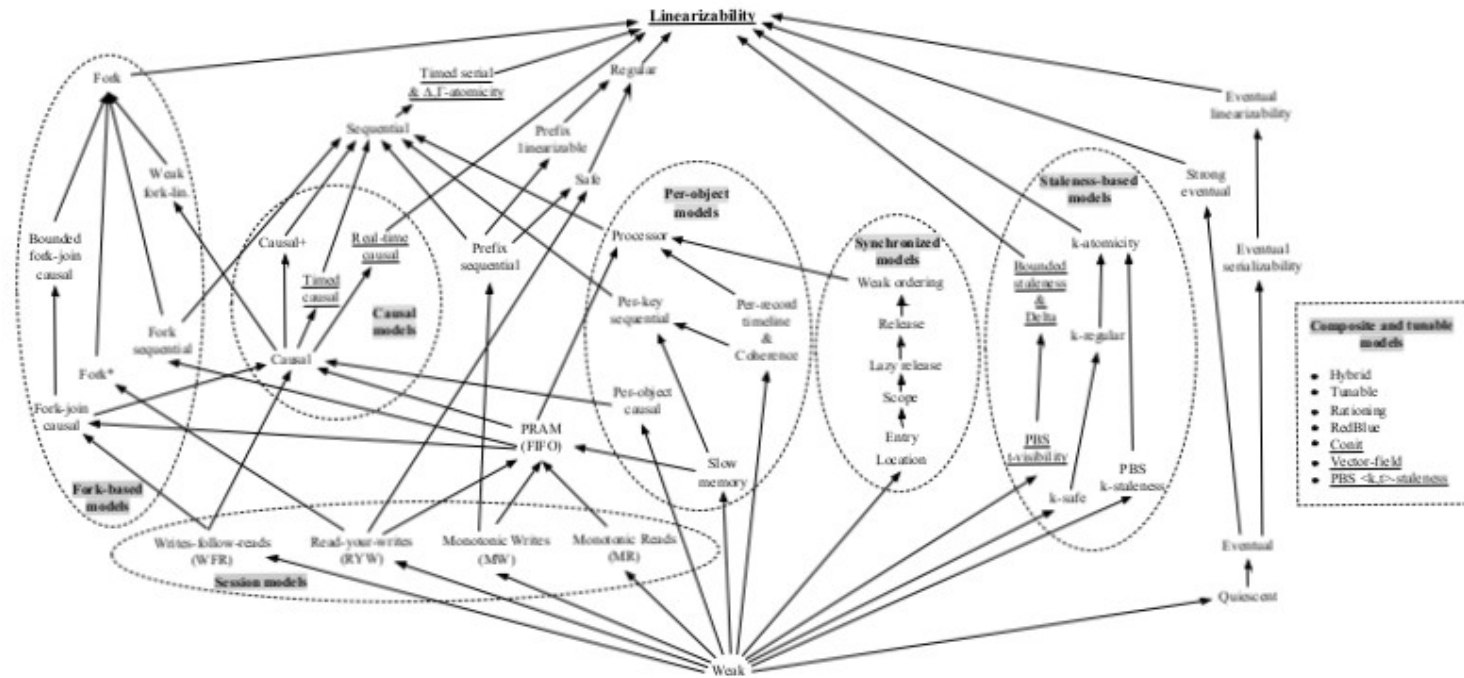


Figure 1: Hierarchy of non-transactional consistency models.

A directed edge from consistency semantics A to consistency semantics B means that any execution that satisfies B also satisfies A. Underlined models explicitly reason about timing guarantees.

# Asynchronous Replication and Eventual Consistency

(AP of CAP. For synchronous replication see consensus protocols like Paxos, Zab, Raft etc.)

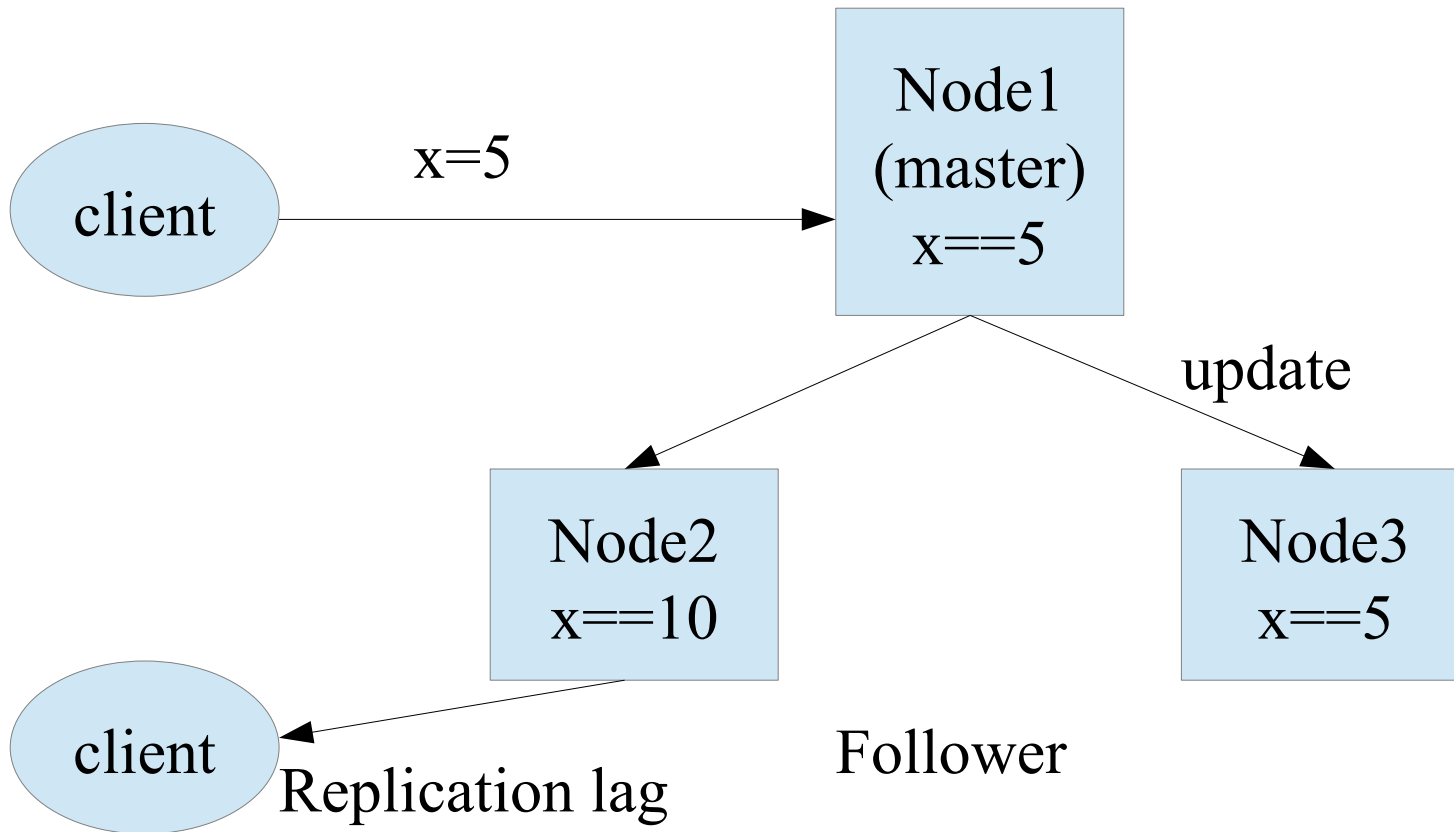
# Replication Models

1. who does the update? Single Master or multiple masters
2. What is updated? State transfer or operation transfer?
3. How are updates ordered?
4. How are conflicts handled/detected?
5. How are updates propagated to replica nodes?
6. What does the system guarantee with respect to divergence?

**Roughly after [Saito]**

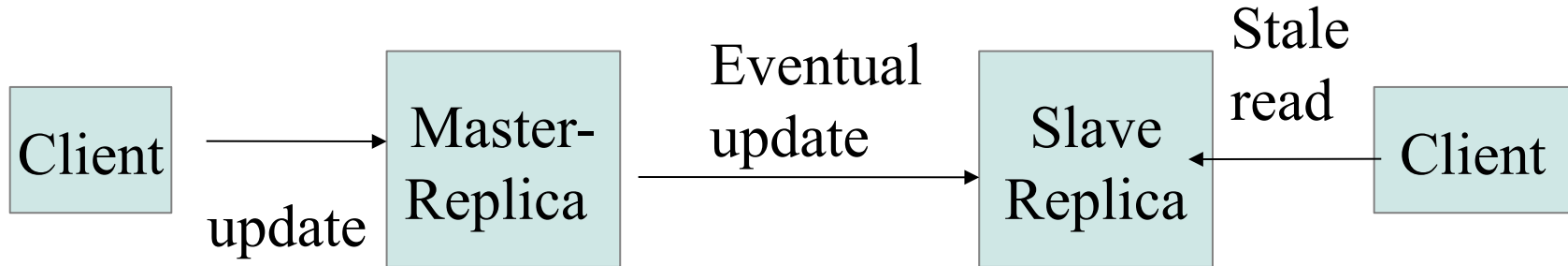


# Single-Leader Replication



Single-Master replication has many advantages: ordered updates, efficient caching, highly available reads. Problems are: how far behind are replicas? What happens, when leader crashes? How long until followers take over? (lease problem). Not good for primary keys and other critical resources. Some systems offer different API QoS levels.

# Eventually consistent reads



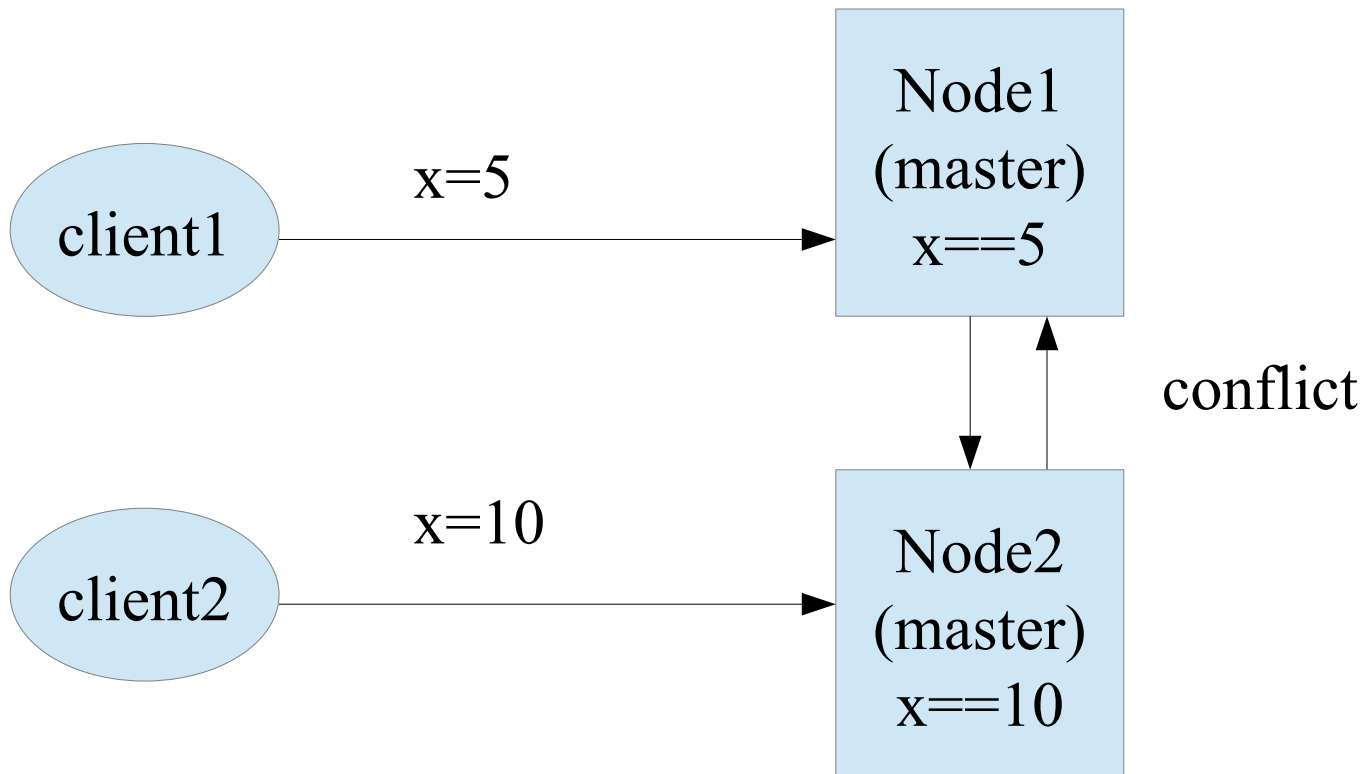
API Examples:

PNUTS (Yahoo): read-any, read-critical, read-latest, write, test-and-set-write.

SimpleDB, GoogleAE: read\_consistent, read\_eventually\_consistent

“The data returned by a read operation is the value of the object at some past point in time but not necessarily the latest value” (Doug Terry, MSR Technical Report October 2011)

# Multi-Master Replication



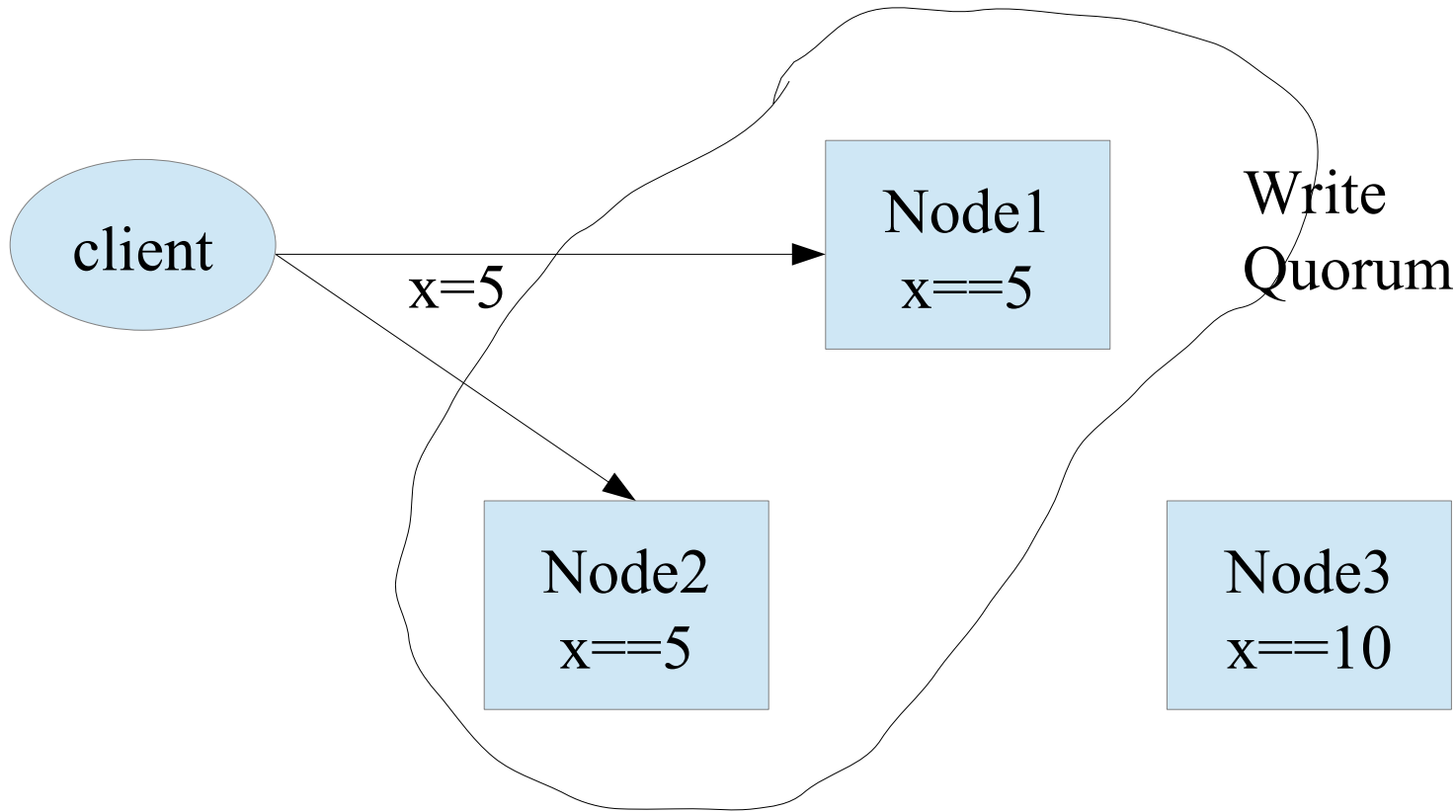
Update conflict: two identical rows changed on two servers

Uniqueness conflict: two identical primary (unique) keys added in same table on two servers

Delete conflict: during delete of a row the same row is changed on a different server.

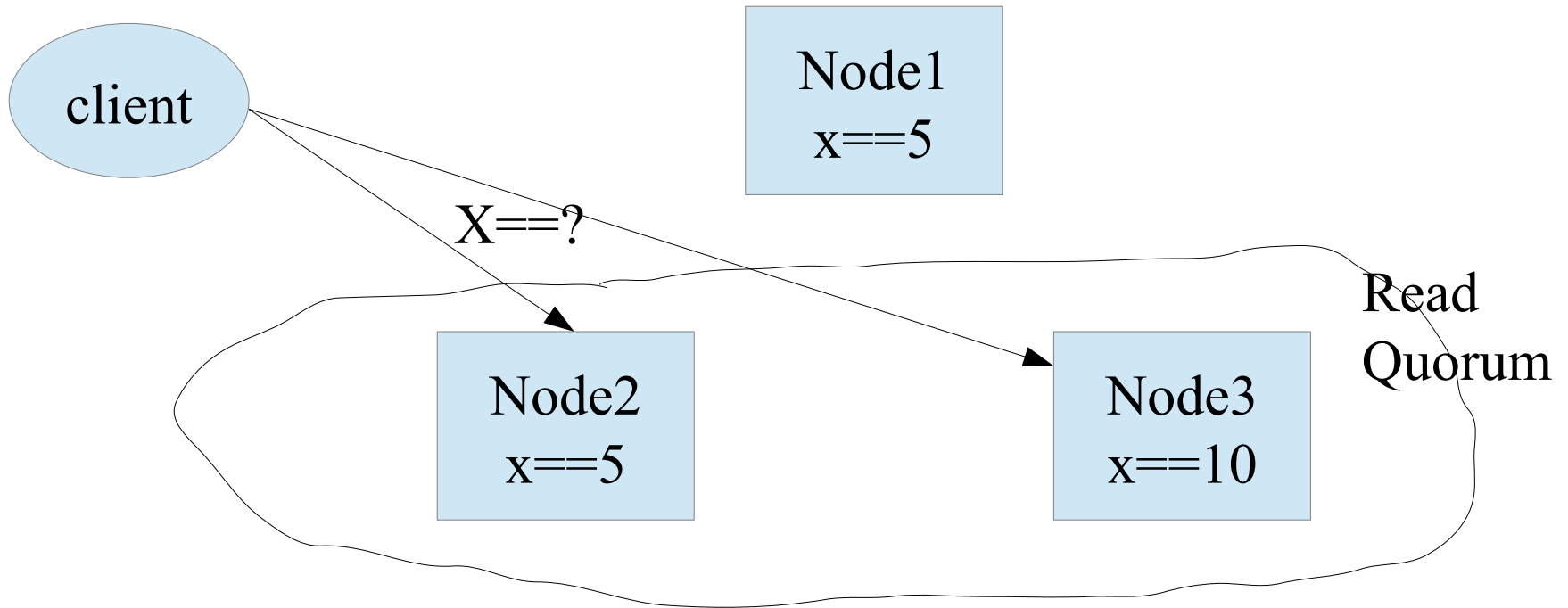
Multi-leader systems typically offer several conflict resolution strategies (last writer wins, keeping different versions, anti-entropy background merge/resolve)

# Leader-less (Quorum) Replication: Write



Without a leader the client decides on how many machines to write to or read from. The formula in general is:  $W+R>N$ , where  $N$  is the number of machines in the replication group. Clients can use the formula to either speed up writes or reads, because quorum systems in general suffer from long tail effects. If a quorum is not available, the client can write to a “sloppy quorum” and risk the write getting lost. Without anti-entropy there is a high danger of partial writes in the system, which are not cleaned up. (See Kleppmann)

# Leader-less (Quorum) Replication: Read



Some systems detect inconsistencies during read. They can either automatically perform a cleanup (e.g. use version number to hand back the correct  $x$  from Node2), or offer both values for the client to decide. (see Kleppmann)

# Session Modes of Asynchronous Replication

## Session Guaranties with optimistic replication

“**Read your writes**” (RYW) guarantees that the contents read from a replica incorporate previous writes by the same user.

“**Monotonic reads**” (MR) guarantees that successive reads by the same user return increasingly up-to-date contents.

“**Writes follow reads**” (WFR) guarantees that a write operation is accepted only after writes observed by previous reads by the same user are incorporated in the same replica. No jumping back in time with a replica that missed some writes.

“**Monotonic writes**” (MW) guarantees that a write operation is accepted only after all write operations made by the same user are incorporated in the same replica. (read set of client received from replica will show those write events)

**After [Saito]. Remember that it is transparent to the client which replica answers a request**

These guarantees seem to enable “sequential-consistency” for a specific client. In other words: Program order of updates from this client is respected by the system. Clients can track those guarantees, e.g. with vector clocks

# Session Anomalies

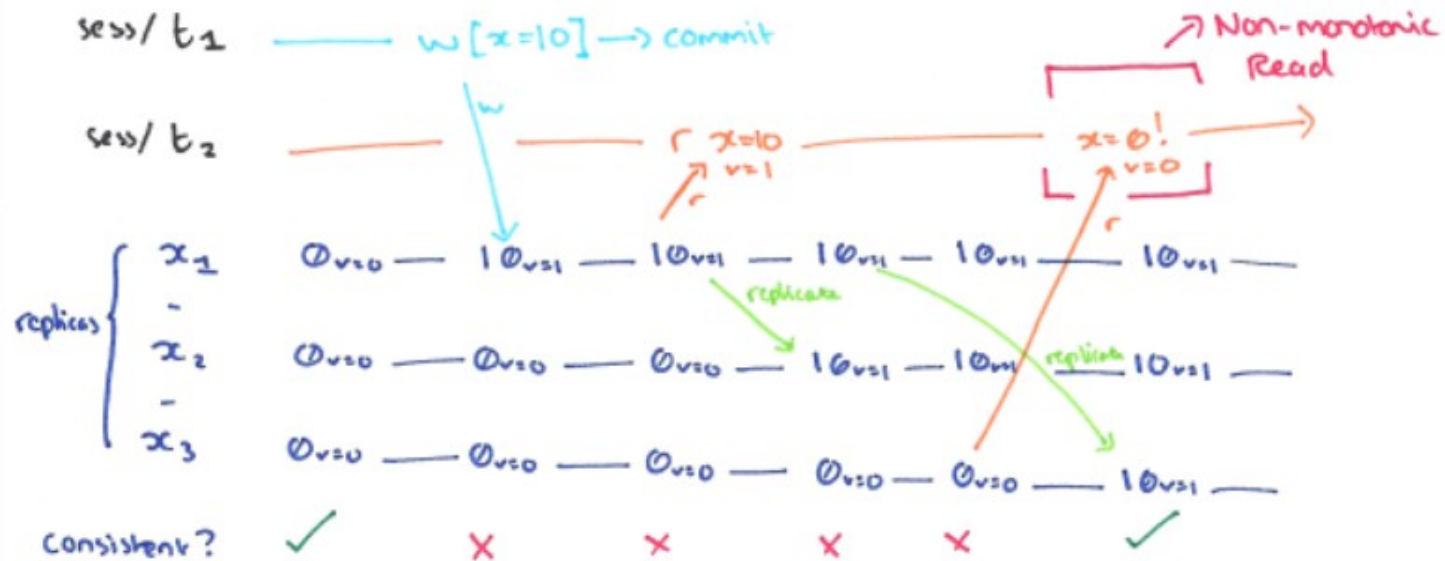
- non-monotonic reads
- non-monotonic writes
- non-monotonic transactions
- not-reading-my-writes

After: Adrain, Coyler, The Morning Paper

<https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/>

# Non-Monotonic Reads

A *non-monotonic read* anomaly occurs within a session when a read of some object returns a given version, and a subsequent read of the same object returns an earlier version. The *Monotonic Reads* guarantee prevents these anomalies and ensures that reads from each item progress according to a total order.



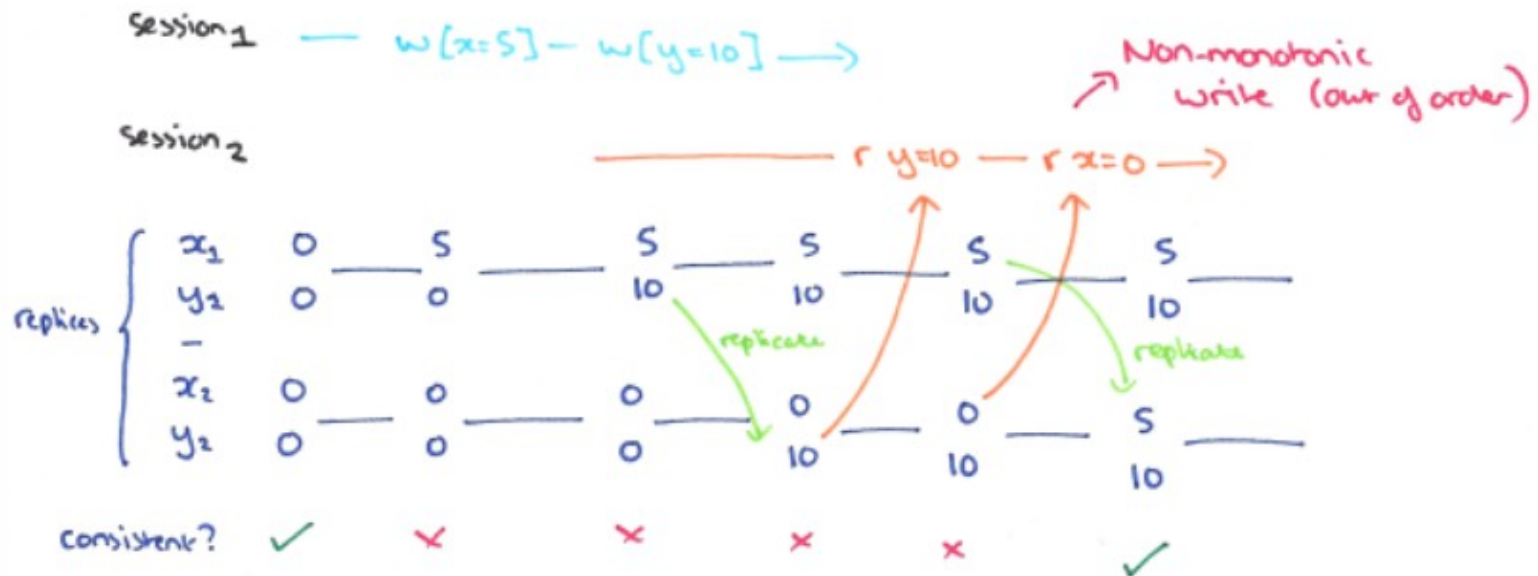
After: Adrain, Coyle, The Morning Paper

<https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/>



# Non-Monotonic Writes

A *non-monotonic write* anomaly occurs if a session's writes become visible out of order. The *Monotonic Writes* guarantee prevents these anomalies and ensures that all writes within a session are made visible in order.

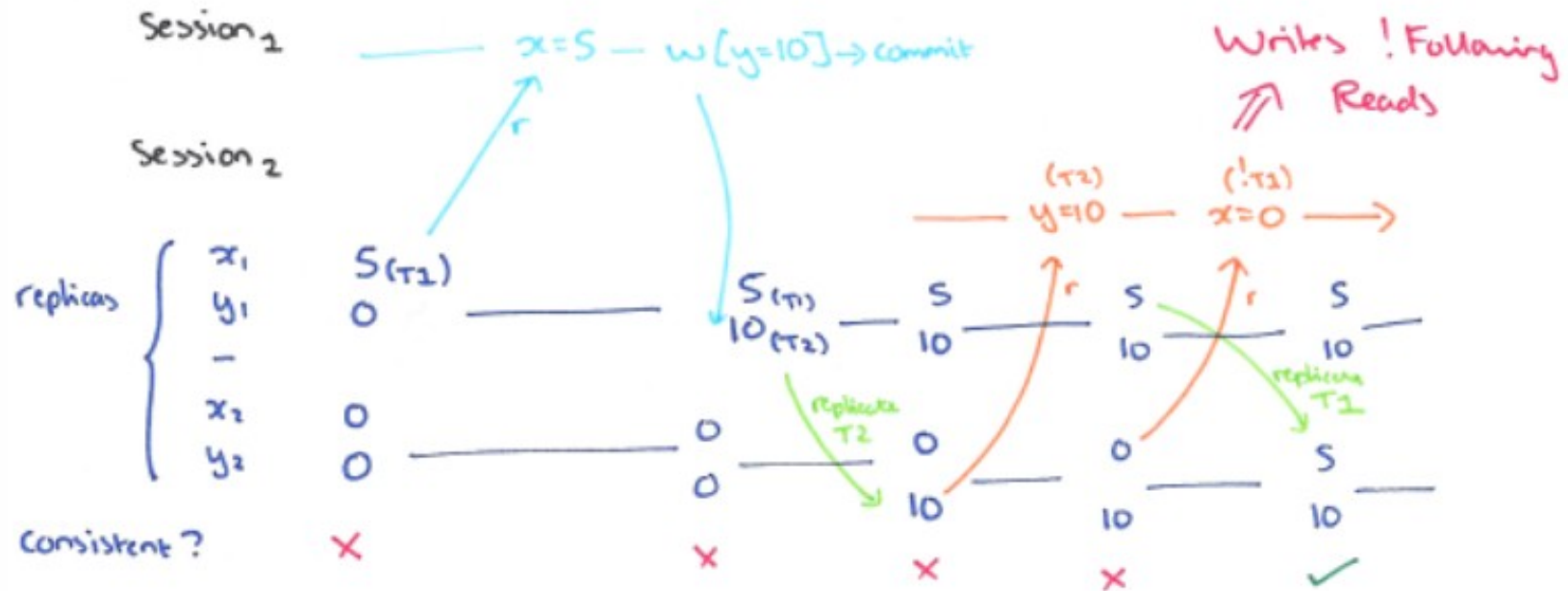


After: Adrain, Coyler, The Morning Paper

<https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/>

# Non-Monotonic Transactions

A *non-monotonic transaction* anomaly occurs if a session observes the effect of transaction T1 and then commits T2, and another session sees the effects of T2 but not T1. The *Writes Follow Reads* guarantee prevents these anomalies.

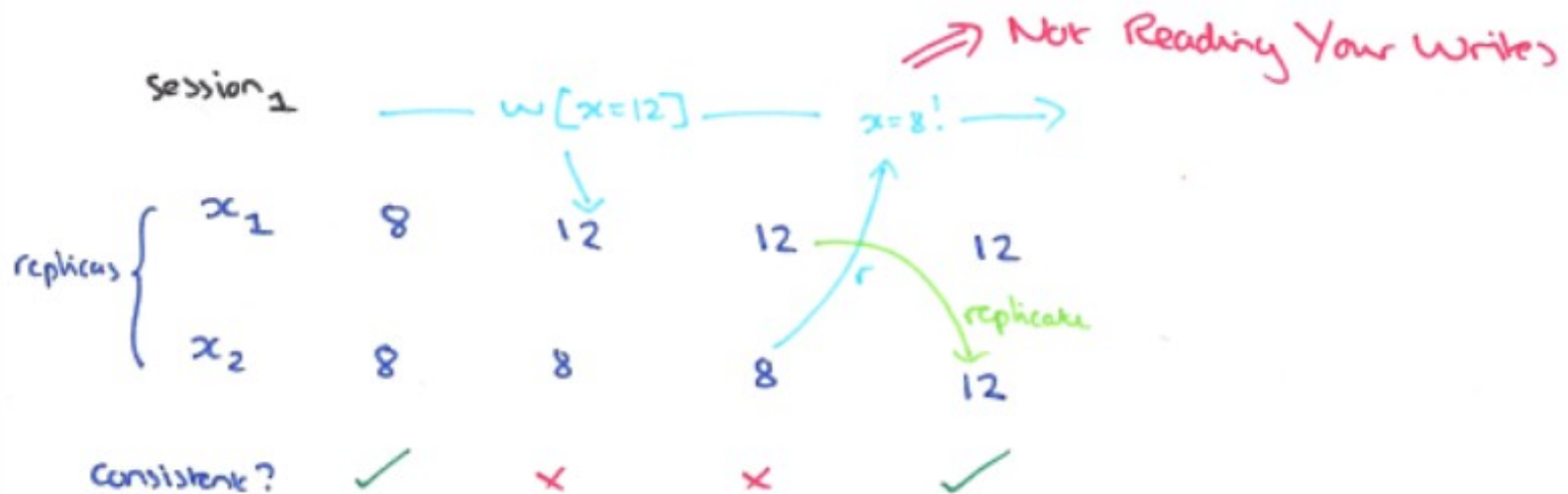


After: Adrain, Coyle, The Morning Paper

<https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/>

# Not-Reading-My-Writes

The *Read Your Writes* guarantee ensures that whenever a client reads a value after updating it, it sees that value (or one installed after it). As far as I know, the anomaly that arises when you do not read your writes does not have a name. Perhaps we could call the failure to read your writes an *arrest anomaly* 😊.



After: Adrain, Coyle, The Morning Paper

<https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/>

# Global Modes of Replication

Strong Consistency	See all previous writes	Ordered, real, monotonic, complete
Consistent Prefix	Ordered sequence of writes/snapshot isolation	Ordered, real, x latest missing, snapshot isolation like
Bounded Staleness	See all writes older than x or every write except the last y	Ordered, real, x latest missing, monotonic increasing due to bound
Eventual Consistency	See subset of prev. writes	Unordered, un-real, incomplete

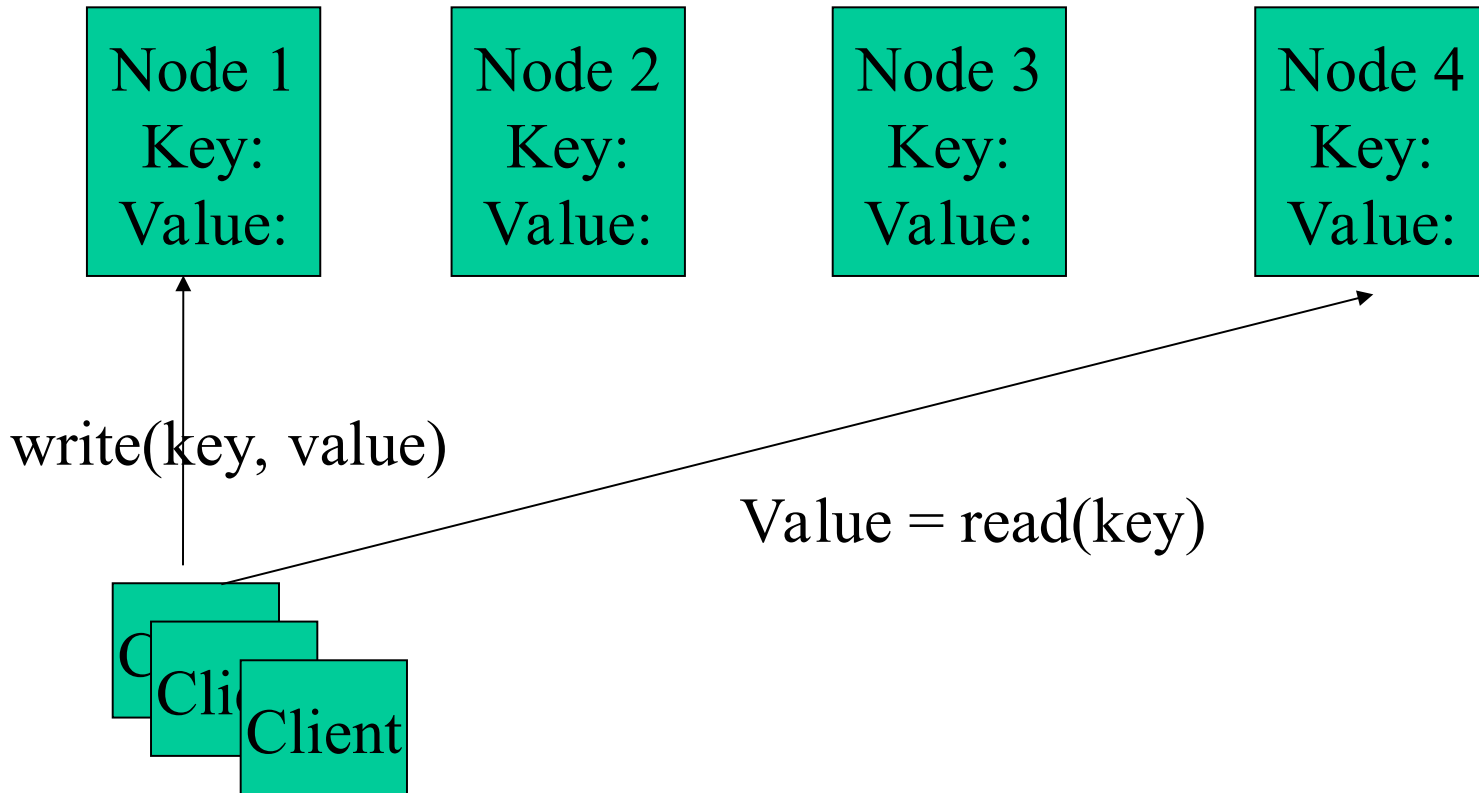
# Trade-Offs

Guarantee	Consistency	Performance	Availability
Strong Consistency	excellent	poor	poor
Eventual Consistency	poor	excellent	excellent
Consistent Prefix	okay	good	excellent
Bounded Staleness	good	okay	poor
Monotonic Reads	okay	good	good
Read My Writes	okay	okay	okay

**Table 2. Consistency, Performance, and Availability Trade-offs**

Taken from D. Terry. Can you explain the difference between CP and BS with respect to the three properties?

# Example: Replicated Key-Value Store



Depending on the consistency guarantees offered by the store, clients see very different results, e.g. because they read from a different replica each time.

# Exercise: Soccer Game

Your job: Given a soccer game, you need to determine the right consistency guarantees needed for each role in the game.

See: owncloud: game.odt

Google drive: ULS: game.odt or game.pdf

# Resources (1)

- Nancy Lynch, Distributed System Algorithms (mathematical proofs of synchronous and asynchronous DS algorithms)
- Ken Birman, Reliable Distributed Systems
- Paxos made live (google)
- <https://github.com/benjaminanweihao/distributed-systems-references>(nice collection of important papers)
- Brian Keating, Challenges Involved in Multimaster Replication, [www.dbspecialists.com/files/presentations/mm\\_replications.html](http://www.dbspecialists.com/files/presentations/mm_replications.html)
- Roberto Baldoni et.al, Vector Clocks [http://net.pku.edu.cn/~course/cs501/2008/reading/a\\_tour\\_vc.html](http://net.pku.edu.cn/~course/cs501/2008/reading/a_tour_vc.html)
- **J. Welch, CSCE 668 DISTRIBUTED ALGORITHMS AND SYSTEMS, <https://parasol.tamu.edu/people/welch/teaching/668.f11/set10-consensus2.ppt> (byzantine failures)**
- Replicated Data Consistency Explained Through Baseball, Doug Terry, Microsoft. MSR Technical Report October 2011



# Resources (2)

Ken Birman on FLP,

<http://www.cs.cornell.edu/courses/cs614/2002sp/cs614%200--%20FLP.ppt>

- [Vogels] Werner Vogels, Eventually Consistent – Revisited,  
[http://www.allthingsdistributed.com/2008/12/eventually\\_consistent.html](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html)
- Sam Toueg, Failure Detectors – A Perspective
- Optimistic Replication, YASUSHI SAITO, MARC SHAPIRO, MS-Research
- Rajagopal Subramaniyan et.al., GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems
- Scalable Management for Global Services, Ken Birman

# On CAP/Critique

- Brewer. Towards robust distributed systems. Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (2000) vol. 19 pp. 7—10
- Gilbert and Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News (2002) vol. 33 (2) pp. 59
- DeCandia et al. Dynamo: Amazon’s highly available key-value store. SOSP ‘07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (2007)
- Fox and Brewer. Harvest, yield, and scalable tolerant systems. Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on (1999) pp. 174—178
- Brewer. Lessons from giant-scale services. Internet Computing, IEEE (2001) vol. 5 (4) pp. 46 – 55
- Martin Kleppmann, Please stop calling databases CP or AP,  
<https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html>
- Coda Hale, You Can’t Sacrifice Partition Tolerance,  
<http://codahale.com/you-cant-sacrifice-partition-tolerance/#ft2> (I took most references from his posting)
- S.Gilbert, Nancy A. Lynch, Perspectives on the CAP Theorem,  
<https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf> (very good intro)

# On Coordination/Atomic Broadcast

ZooKeeper: A Distributed Coordination Service for Distributed Applications,  
<http://zookeeper.apache.org/doc/trunk/zookeeperOver.html>

ZooKeeper's atomic broadcast protocol: Theory and practice, Andre Medeiros, March 20, 2012

Marco Serafini, Zab vs. Paxos,

Coordination Avoidance in Distributed Databases, By Peter David Bailis, (dissertation)

Jessica Kerr, Provenance and Causality in Distributed Systems,  
<http://blog.jessitron.com/2016/09/provenance-and-causality-in-distributed.html>

# General

Alvaro Videla, What we talk about when we talk about Distributed Systems,  
<http://videlalvaro.github.io/2015/12/learning-about-distributed-systems.html>

<http://bravenewgeek.com/from-the-ground-up-reasoning-about-distributed-systems-in-the-real-world/>

- lists important papers and explains things from the bottom up

<https://www.cse.unsw.edu.au/~cs9243/16s1/papers/fallacies.pdf>

<http://blog.empathybox.com/post/19574936361/getting-real-about-distributed-system-reliability-> questions the idea that failures in DS are independent. This invalidates the redundancy argument a bit and shows the importance of monitoring and system management.